INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

AUTOMATA FOR LEARNING SEQUENTIAL TASKS

DISSERTATION

**Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of the Ohio State University**

by

Chee Hung Ben CHOI, B.S, M.S.

The Ohio State University

1996

Dissertation Committee:

**Prof. Kenneth J. Breeding, Adviser
Prof. Steven B. Bibyk
Prof. Charles A. Klein**

Approved by

Adviser

Department of Electrical Engineering

UMI Number: 9630870

**Copyright 1996 by
Choi, Chee Hung Ben**

All rights reserved.

**UMI Microform 9630870
Copyright 1996, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Copyright by
Chee Hung Ben CHOI
1996

ABSTRACT

This dissertation describes a system that is capable of learning both combinational and sequential tasks. The system learns from sequences of input/output examples in which each pair of input and output represents a step in a task. The system uses finite state machines to represent learned models. The resultant learned models are often deterministic finite state machines which often contain minimal number of states. Thus, the resultant learned models can easily and economically be implemented in VLSI. On the contrary, neural network models are more difficult to be implemented in hardware. This dissertation proposes a method for inferring finite state machines from sequences of input/output examples. The system is often capable of inferring a reasonable finite state machine from just one sequence of examples. On the contrary, recurrent neural networks often require thousands of sequences. New algorithms are developed for modifying the finite state machines to allow the system to learn new information when additional examples are available. In addition, new algorithms are developed to allow the system to handle inconsistent information that may result from noise in the training examples. A simulation program is developed to test the system and several application examples are provided.

Dedicated to my Parents

ACKNOWLEDGMENTS

I express my deepest appreciation to my Parents for their love, care, and support, without which my pursuit of my endeavors would not be possible.

I wish to thank Professor Kenneth J. Breeding for being my best teacher, my advisor, and my friend. He has been my mentor since I was an undergraduate.

I thank Professor Steven B. Bibyk and Professor Charles A. Klein for being my dissertation committee members and for their comments and suggestions.

VITA

1986 B.S. Electrical Engineering, The Ohio State University.

1986-1987 .. Research Associate, Engineering Graphic Dept.

1988 M.S. Electrical Engineering, The Ohio State University.

1988-1989 .. Computer Consultant, Academic Computing Services.

1988-1990 .. Research Associate, Computer Information Science Dept.

PUBLICATIONS

“Generic Associative Memory for Information Retrieval”, IWDM Database Machines, pp. 202-214, 1989.

“Two Papers on Text-Retrieval and AI Machines”, Technical Research Report, OSU-CISRC-3/89-RT12, Department of Computer Information Science, The Ohio State University, 1989.

“Parallel Distributed Computer Architecture and General Associative Memory for Artificial Intelligent Processing”, M.S. Thesis, The Ohio State University, 1988.

“Text to Fingerspelling and Speech on the Amiga Micro-computer”, IEEE Ninth Annual Conference of the Engineering in Medicine and Biology Society, pp. 1769-1770, 1987.

FIELDS OF STUDY

Major Field: Electrical Engineering: Computer Engineering

Minor Field: Computer and Information Science

Minor Field: Solid State Microelectronics

TABLE OF CONTENTS

Abstract.....	ii
Dedication	iii
Acknowledgments	iv
Vita	v
List of Figures.....	x

CHAPTERS:

1. INTRODUCTION.....	1
1.1. INTRODUCTION.....	1
1.2. APPROACH AND CONTRIBUTIONS OF THIS INVESTIGATION.....	3
1.3. ORGANIZATION	6
2. RELATED WORK.....	9
2.1. INTRODUCTION.....	9
2.2. NEURAL NETWORK APPROACH	9
2.2.1. <i>Introduction</i>	9
2.2.2. <i>Comparison with neural networks</i>	10
2.3. ARTIFICIAL INTELLIGENT APPROACH.....	15
2.3.1. <i>Introduction</i>	15
2.3.2. <i>Learning systems using AI approach</i>	16
2.3.3. <i>Comparison with related systems</i>	18
3. OVERVIEW OF THE SYSTEM FOR LEARNING SEQUENTIAL TASKS.....	24
3.1. INTRODUCTION.....	24
3.2. INTERNAL MODELS	24
3.3. TRAINING DATA.....	26
3.4. LEARNING ALGORITHMS	27

4. CREATING A NEW INTERNAL MODEL.....	29
4.1. INTRODUCTION.....	29
4.2. PROCESS FOR CREATING A NEW INTERNAL MODEL.....	29
4.2.1. <i>Converting sequence of input/output examples</i>	31
4.2.2. <i>Checking compatible</i>	36
4.2.3. <i>Finding maximum compatibility classes</i>	39
4.2.4. <i>Deriving all prime compatibility classes</i>	42
4.2.5. <i>Finding a minimum closed cover</i>	46
4.2.6. <i>Deriving a minimum finite state machine</i>	52
5. CHECKING CONSISTENCY AND UPDATING INTERNAL MODELS.....	58
5.1. INTRODUCTION.....	58
5.2. PROCESS FOR CHECKING CONSISTENCY AND UPDATING INTERNAL MODEL	58
5.3. ADDING NEW INPUT COLUMNS	63
5.4. CHECKING CONSISTENCY	64
5.5. COMPARING OUTPUT SEQUENCES AND SPECIFYING DON'T CARES	66
5.6. SEARCHING AND BACKTRACKING	69
5.7. REMOVING POSSIBLE NEXT STATES	71
6. HANDLING INCONSISTENT INFORMATION	77
6.1. INTRODUCTION.....	77
6.2. PROCESS FOR HANDLING INCONSISTENT INFORMATION.....	78
7. TRAINING EXAMPLES AND SIMULATION RESULTS.....	81
7.1. INTRODUCTION.....	81
7.2. TRAINING THE SYSTEM	81
7.3. A PARITY CHECKER	83
7.4. AN UP-DOWN COUNTER	85
7.5. A SEQUENCE DETECTOR	87
7.6. A TASK INVOLVING MANY STATES	96
7.7. A SIMPLE CODE BREAKER	100

8. PERFORMANCE	109
8.1. INTRODUCTION.....	109
8.2. PERFORMANCE OF THE PROCESS FOR CREATING A NEW INTERNAL MODEL	109
8.3. PERFORMANCE OF THE PROCESS FOR CHECKING CONSISTENCY AND UPDATING INTERNAL MODELS .	111
9. SUMMARY AND CONCLUSION.....	115
10. REFERENCES.....	118

LIST OF FIGURES

FIGURE 2.2.1. STRUCTURE OF RECURRENT NEURAL NETWORK [29]. ...	12
FIGURE 2.3.1. LEARNING FROM OUTPUT SEQUENCE [55].....	19
FIGURE 2.3.2. BUILDING A STATE DIAGRAM FORM INPUT SEQUENCES AND THEIR CLASSES [12].....	20
FIGURE 3.2.1. EXAMPLE OF AN INTERNAL MODEL.....	25
FIGURE 3.3.1. A SEQUENCE OF INPUT/OUTPUT EXAMPLES.....	26
FIGURE 3.4.1. OVERVIEW OF THE LEARNING PROCESS.	27
FIGURE 4.2.1. STEPS FOR CREATING A NEW INTERNAL MODEL.	30
FIGURE 4.2.2. INPUT/OUTPUT SEQUENCE.....	32
FIGURE 4.2.3. INCOMPLETELY SPECIFIED STATE TABLE	33
FIGURE 4.2.4. A MODIFIED IMPLICATION TABLE.	38
FIGURE 4.2.5. FINDING MAXIMUM COMPATIBILITY CLASSES.	40
FIGURE 4.2.6. MAXIMUM COMPATIBILITY CLASSES AND THEIR IMPLIED CLASSES.	43
FIGURE 4.2.7. ALL PRIME COMPATIBILITY CLASSES.....	45
FIGURE 4.2.8. SEARCH TREE FOR FINDING MINIMUM CLOSED COVER...	48
FIGURE 4.2.9. RESULT OF STATE REDUCTION.	53
FIGURE 5.2.1. CHECKING CONSISTENCY AND UPDATING INTERNAL MODEL.	59
FIGURE 5.2.2. EXAMPLE FOR UPDATING INTERNAL MODEL.	60
FIGURE 5.5.1. EXAMPLE FOR COMPARING TWO SEQUENCES.....	66
FIGURE 5.6.1. TRANSFORM SEQUENCES INTO A SEARCH TREE.....	69
FIGURE 5.7.1. EXAMPLE FOR REMOVING POSSIBLE NEXT STATES.	72
FIGURE 5.7.2. CHOOSING POSSIBLE STATES TO BE REMOVED.	74
FIGURE 6.2.1. PROCESS FOR HANDLING INCONSISTENT INFORMATION.	79

FIGURE 7.3.1. A SCREEN SHOT OF THE SIMULATION PROGRAM.	83
FIGURE 7.3.2. PARITY CHECKER: I/O SEQUENCE.	84
FIGURE 7.3.3. PARITY CHECKER: RESULTANT STATE TABLE.....	84
FIGURE 7.4.1. COUNTER: I/O SEQUENCE.....	85
FIGURE 7.4.2. COUNTER: RESULTANT STATE TABLE.	86
FIGURE 7.4.3. COUNTER: TEST RESULT.	87
FIGURE 7.5.1. SEQUENCE DETECTOR: I/O SEQUENCE 1.....	88
FIGURE 7.5.2. SEQUENCE DETECTOR: STATE TABLE 0.....	88
FIGURE 7.5.3. SEQUENCE DETECTOR: TEST RESULT 1.	89
FIGURE 7.5.4. SEQUENCE DETECTOR: I/O SEQUENCE 2.....	90
FIGURE 7.5.5. SEQUENCE DETECTOR: RESULT OF SEQUENCE 2.....	90
FIGURE 7.5.6. SEQUENCE DETECTOR: STATE TABLE 0 AFTER SEQUENCE 2.	91
FIGURE 7.5.7. SEQUENCE DETECTOR: I/O SEQUENCE 3.....	91
FIGURE 7.5.8. SEQUENCE DETECTOR: STATE TABLE 1.....	92
FIGURE 7.5.9. SEQUENCE DETECTOR: I/O SEQUENCE 4.....	92
FIGURE 7.5.10. SEQUENCE DETECTOR: STATE TABLE 2.....	93
FIGURE 7.5.11. SEQUENCE DETECTOR: RESULTANT STATE TABLE 0. ...	94
FIGURE 7.5.12. SEQUENCE DETECTOR: RESULTANT STATE TABLE 1. ...	94
FIGURE 7.5.13. SEQUENCE DETECTOR: RESULTANT STATE TABLE 2. ...	94
FIGURE 7.5.14. SEQUENCE DETECTOR: RESULTANT TEST 0.....	95
FIGURE 7.5.15. SEQUENCE DETECTOR: RESULTANT TEST 1.....	95
FIGURE 7.5.16. SEQUENCE DETECTOR: RESULTANT TEST 2.....	95
FIGURE 7.6.1. MANY STATES: I/O SEQUENCE 1.	96
FIGURE 7.6.2. MANY STATES: STATE TABLE 0.....	97
FIGURE 7.6.3. MANY STATES: I/O SEQUENCE 2.	98
FIGURE 7.6.4. MANY STATES: STATE TABLE 1.....	99
FIGURE 7.7.1. CODE BREAKER: MESSAGE 1.....	101
FIGURE 7.7.2. CODE BREAKER: STATE TABLE 0.	101
FIGURE 7.7.3. CODE BREAKER: MESSAGE 2.	101
FIGURE 7.7.4. CODE BREAKER: STATE TABLE 1.	102
FIGURE 7.7.5. CODE BREAKER: MESSAGE 3.	103
FIGURE 7.7.6. CODE BREAKER: STATE TABLE 2.....	104
FIGURE 7.7.7. CODE BREAKER: RESULTANT STATE TABLE 0.....	104
FIGURE 7.7.8. CODE BREAKER: RESULTANT STATE TABLE 1.....	104
FIGURE 7.7.9. CODE BREAKER: RESULTANT STATE TABLE 2.....	104
FIGURE 7.7.10. CODE BREAKER: COMPARISON 0.	106

FIGURE 7.7.11. CODE BREAKER: COMPARISON 1.	106
FIGURE 7.7.12. CODE BREAKER: COMPARISON 2.	106
FIGURE 7.7.13. CODE BREAKER: TEST RESULT 0.....	107
FIGURE 7.7.14. CODE BREAKER: TEST RESULT 1.....	107
FIGURE 7.7.15. CODE BREAKER: TEST RESULT 2.....	107
FIGURE 8.2.1. SUMMARY OF SIMULATION RESULTS FOR CREATING NEW INTERNAL MODELS.....	111
FIGURE 8.3.1. REDUCE SEARCH SPACE BY DEPTH-FIRST SEARCH PROCESS.	112
FIGURE 8.3.2. SUMMARY OF SIMULATION RESULTS FOR CHECKING AND UPDATING INTERNAL MODELS.	114

1. INTRODUCTION

1.1. Introduction

A machine that can learn is generally defined as a machine that can improve its own performance. Using such a broad definition, many types of research can be classified into machine learning [14][43][42]. One type of machine learning, that of learning from examples [43], has received much attention from researchers in neural networks [56][38][37][58] and in artificial intelligence [17]. It is also the topic of this investigation.

This investigation addresses the problem of learning sequential tasks from examples.

To learn, a machine has to possess the ability to memorize and has to be able to perform inference [43].

Definition 1.1. *Learning* is defined in this investigation as inferring a general model from examples.

Definition 1.2. A task is specified by a sequence of steps. Each step consists of an input and an output, a stimulus and a response, or a condition and an action. A task is called a *sequential task* if the output of a step depends on some previous steps. It is called a *combinational task* if the output of any step does not depend on any other steps.

A usual way to teach or to train a machine is to provide the machine with examples. In this investigation, examples are provided in the format of input/output sequences.

Definition 1.3. An *input/output sequence* (also called a *sequence of input/output examples*) is a finite sequence of ordered pairs, each of which consists of an input and an output. That is

$$((x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)).$$

Each (x_k, y_k) , for $0 \leq k \leq n$, is an ordered pair, in which x_k is an input and y_k is an output. The collection of all inputs, $(x_0, x_1, \dots, x_n)$, is

called an *input sequence*. The collection of all outputs, $(y_0, y_1, \dots, y_n)$, is called an *output sequence*.

This investigation addresses the following three problems:

- (1) how to infer a general model from a sequence of input/output examples,
- (2) how to modify the existing model to accommodate new information provided by additional sequences of input/output examples, and
- (3) how to handle inconsistent information that may result from noise or from trainers who provide inconsistent examples.

1.2. Approach and contributions of this investigation

This investigation proposes to use finite state machines, in particular Mealy machines, to represent learned information.

Definition 1.4. A *finite state machine (a Mealy machine)* is defined with 6-tuple [32][15]:

$$(I, O, S, s_0, \delta, \lambda)$$

where, I is a set of input symbols; O is a set of output symbols; and S is a set of states. A state s_0 in S is called an initial state that is the starting state of the machine. δ is called a state transition function that maps from the cross product of I and S to the power set of S , that is,

$$\delta : I \times S \rightarrow \rho(S)$$

where, $\rho(S)$ is the power set of S , i.e., $\rho(S) = \{X \mid X \subseteq S\}$. λ is called an output function that maps from the cross product of I and S to O , that is,

$$\lambda : I \times S \rightarrow O$$

A system is developed in this investigation to address the three problems outlined earlier.

The approach and contributions of this investigation are outlined as shown below:

1. Using algorithms developed by the author and existing algorithms, a process is developed to allow the system to learn new tasks (described in Chapter 4). The process is able to infer a minimum finite state machine from any sequence of input/output examples. Although several existing algorithms are used, the application of the process as a tool for inference in a learning system is new.
2. New algorithms are developed by the author for modifying the finite state machines to allow the system to accommodate new information (described in Chapter 5). A new concept is defined for checking consistency between a sequence of input/output examples and a finite state machine. New algorithms are used to restrict or to expand the scope of the existing machines in order to accommodate new information provided by additional sequences of input/output examples.

3. New algorithms are developed by the author to allow the system to handle inconsistent information that may result from noise or from trainers who provide inconsistent examples (described in Chapter 6). The system records inconsistent information by creating new finite state machines and by weighting existing finite state machines.

1.3. Organization

This dissertation is organized into nine chapters.

Chapter 1, this chapter, provides an overview of the research problem, approach, and contributions of this investigation.

Chapter 2 classifies related work into neural network approach and artificial intelligent approach. It also provides comparisons of this investigation to related work found in both the approaches.

Chapter 3 provides an overview of the system developed in this investigation. It shows how the system represents internal models and training data. It outlines the learning algorithms developed in this investigation.

Chapter 4 describes the process for creating a new internal model. The process is able to infer a minimum finite state machine from a sequence of input/output examples.

Chapter 5 describes the process for checking consistency and updating internal models. The process is able to detect any inconsistency between a sequence of input/output examples and an internal model. It is able to expand or to limit the scope of the internal models to conform to new information provided by additional sequences of input/output examples.

Chapter 6 describes the process for handling inconsistent information. This process uses the process for checking consistency to detect any inconsistency that may result from noise. It uses the process for creating a new internal model to build new internal models to record

any inconsistency or any abrupt change. It also evaluates each of the internal models by assigning a weight for each of the models.

Chapter 7 describes how to train the system to perform certain tasks or functions. It also provides several application examples and their simulation results.

Chapter 8 evaluates performance of the learning system. It describes the methods implemented in the system for improving performance.

Chapter 9 provides summary and conclusion of this investigation.

2. RELATED WORK

2.1. Introduction

In this chapter, research for addressing the problem of learning from examples is classified into: neural network approach and artificial intelligent approach. The system developed in this investigation is compared to related systems found in both the approaches.

2.2. Neural network approach

2.2.1. Introduction

Systems using a neural network approach utilize network topologies as the internal models to represent the learned information. The learned information is embedded in the nodes and in the connections of the networks. Each node can perform certain simple functions. The

systems can change the functions performed by each node, and can also change the connections between the nodes.

The objective of the learning systems is to use training methods to change the networks in such a way that the outputs produced by the network match as closely as possible to the required outputs. The inputs and the outputs of the networks are usually real numbers, and they are usually given in pairs consisting of an input and the corresponding required output ([56][57][25][31][2][49][46]).

2.2.2. Comparison with neural networks

The system developed in this investigation will be compared to neural network approach in two aspects: training process and implementation.

2.2.2.1. Training process

Training the system developed in this investigation requires a small number of sequences of examples. The system is often capable of inferring a reasonable finite state machine from just one sequence of

input/output examples. Moreover, the system is able to handle sequential tasks that require many states. The simulation results are provided in Chapter 7.

On the contrary, training recurrent neural networks [20][29] (see Figure 2.2.1) requires a very large number of sequences. For example, it requires approximately 2400 to 409600 sequences of examples to train a recurrent neural network to determine whether an input sequence contains an even or odd number of 1's (Bengio [8][9]). The number of sequences required depends on the length of the input sequence and the training algorithm used. For an input sequence having 10 inputs, for instance, all five training algorithms used in the study by Bengio [8][9] requires more than 6400 training sequences.

On the other hand, recurrent neural networks should perform better in noisy environment than the system developed in this investigation.

In addition, it is difficult to train recurrent neural networks for tasks involving “long-term dependencies” [8][9], that is, tasks that have the value of an output depending on an input several steps prior. There is

Figure 2.2.1. Structure of recurrent neural network [29].

also the problem of local minima such that the networks cannot discover optimal solutions (Bianchini [10]).

Difficulties in training recurrent neural networks are the results of the structure of the networks. As shown in Figure 2.2.1, the structure of recurrent neural networks is similar to that of asynchronous sequential circuits (Jordan [29]). Feedback paths in the networks allow them to function like sequential circuits and thus enable the networks to learn sequential tasks. The feedback paths provide additional computational power compared to that of feed-forward neural networks that have structures similar to combinational circuits. The added feedback paths

also make it very difficult to devise effective learning procedures for recurrent neural networks (Bianchini [10]).

2.2.2.2.Implementation

The system developed in this investigation uses finite state machines to represent learned models. The resultant learned models are often deterministic finite state machines that are represented by using state tables. Moreover, the finite state machines often contain minimal number of states. Thus, the resultant learned models can easily and economically be implemented in VLSI.

On the contrary, implementing neural networks in hardware is more difficult. The difficulty is the result of analog models used in most neural networks. Most neural network models require continuous and signed values for representing weights and internal states (also called activation values). They also require operations such as multiplication, summation, and comparison of the continuous values (Rumelhart [56]).

To efficiently implement the continuous values and operations, analog circuits are required.

Thus, researchers investigate using analog VLSI [11][26][39][40][28] and optical computers for implementing neural networks. Researchers in these areas face a number of technical challenges. Researchers in analog VLSI have to overcome problems of variation in chip lithography, dopant density, instability, and imprecision of component parameters [54]. Researchers in optical computing have to overcome problems of limiting dynamic range, and problems of interfacing bottleneck between electronic and optical components [54].

Other attempts to implement neural networks in hardware include simulating the analog networks in digital computers. Some commercial products advertised as Neurocomputers are in fact simulators running in digital computers [25]. It shall be emphasized that these simulators do not exploit in hardware the parallel and distributed computation power of neural network models [54].

2.3. Artificial intelligent approach

2.3.1. *Introduction*

Artificial Intelligent (AI) approach uses symbolic systems and symbolic manipulations as the basis for machine intelligence [52][17]. There are three aspects for evaluating a learning system using an AI approach: (1) internal models used to represent learned information, (2) training data given to the systems, and (3) training methods used by the systems.

A learning system using an AI approach represents the learned information by using symbolic models, such as, production rules that have a format of “IF condition THEN action”, predicate logic, or finite automata.

The training data given to the systems are symbolic data, such as, condition-action examples, sentence examples, or input and output examples.

The training methods used by the systems are symbolic manipulation methods, such as, partitioning, searching, or generate-and-test. The objective of the learning systems is to manipulate the internal models in such a way that the internal models not only can represent the training data efficiently but also can be used as basis for inferences or generalizations.

2.3.2. Learning systems using AI approach

Some research areas that use an AI approach for developing learning systems are machine learning [43], inductive inference [5], learning automata [45], and adaptive control [16].

Machine learning is a broad research area. A machine that can learn is generally defined as a machine that can improve its own performance. Using such a broad definition, many types of research can be classified into machine learning [14][43].

Inductive inference systems were defined as systems that try to infer general rules from examples (Angluin [5]). The original theory of

inductive inference was provided by Gold [23]. The complexity of inferring general rules from given data was described by Gold [22].

Learning automata systems apply the theories of finite state automata and stochastic automata for machine learning (Narendra [45]). Work on learning automata originated in the 1960s [62]. Learning automata have been applied to pattern recognition and robotics [19][6][47].

Adaptive control systems use algebraic expressions as internal models to represent the learned information [61][16][60][30]. The algebraic expressions are usually linear equations or differential equations. The forms of the algebraic expressions are usually given in advance, but the values of their parameters or coefficients are set as unknown. The objective of the learning system is to adjust the unknown parameters to conform to the training data that are usually a set of real numbers. The training methods used by the systems are usually statistical methods, such as, least squares method [34].

2.3.3. Comparison with related systems

The system developed in this investigation will be compared to related AI systems using three aspects: formats of training data, passing computational burden to teachers, and performing experiments.

2.3.3.1. Formats of training data

The system developed in this investigation is trained by using sequences of input/output examples (see Definition 1.3). Each step in a sequence consists of an input and an output, a stimulus and a response, or a condition and an action. Moreover, steps in a sequence are in time order. From the sequences of input/output examples, the system is able to learn sequential tasks.

On the contrary, several systems reviewed in the following restrict their training data to special formats and their examples are not in time order. Thus they are not able to learn sequential tasks.

The system described by Porat [50] requires the training data to be arranged into strict lexicographic order before being given to the

Figure 2.3.1. Learning from output sequence [55].

system. The system also restricts the output to only one bit representing acceptance (denoted by +) or rejection (denoted by -). For example, using a and b as inputs, some strict lexicographic ordered examples are: -a, +b, +aa, and -ab.

Another system restricts the training data to only output sequences [55]. For example, using the sample output sequence shown in Figure 2.3.1(b), the method reported in Rouvellou [55] is able to generate the state diagram shown in Figure 2.3.1(a).

Another system restricts the training data to input sequences and their associated classes [12]. An input sequence and its associated class is

Training data:
 $\{((a), 1), ((a,a), 2), ((a,b), 1), ((b,a), 2), ((a,b,a), 2), ((a,b,b), 1), ((b,b,a), 1), ((a,a,a,a), 2), ((a,a,b,a), 2)\}$

Figure 2.3.2. Building a state diagram form input sequences and their classes [12].

represented by $((i_1, i_2, i_3, \dots), c)$ where $i_1, i_2, i_3, \dots$ is the input sequence and c is the associated class. For example, a set of the training data is shown in Figure 2.3.2. By using the training data, the method reported in Biermann [12] is able to construct the state diagram shown in Figure 2.3.2. To construct the state diagram, the method reported in Biermann [12] requires that several inputs be associated with one output. On the other hand, the system developed in this investigation allows each input to be associated with an output.

Some other systems can only learn from reward or punishment information [47][33][6]. The systems are first given a set of possible

actions. They initially choose an action at random. During training, the trainers tell the systems whether the action is favorable. Then the systems update the likelihood for producing the action.

2.3.3.2. Passing computational burden to teachers

The system developed in this investigation does not pass computational burden to its teacher. It does not ask questions.

On the contrary, several systems that pass the computational burden to their teachers have been reported in [53][4][3][27][36]. Those systems require their teachers to compute functions, such as, whether two sets of the training data are equivalent. For example, Angluin [4] provided an algorithm to infer a regular set that is described by a deterministic finite state acceptor. The algorithm uses a teacher that can answer membership queries as to whether an example is a member of the regular set. The teacher also provides counter-examples if the algorithm makes an incorrect guess. Ibarra [27] addressed the same problem but only allowed the teacher to answer equivalency queries.

Marron [36] addressed the problem by using one initial example followed by membership queries.

2.3.3.3. Performing experiments

The system developed in this investigation does not perform experiments on its environment. It learns from training data provided by a teacher.

On the contrary, the system developed by Rivest [53] requires experiments to be performed on its environment. It also requires a teacher to determine the equivalency between the experiments. Performing experiments on the environment is similar to performing experiments on an unknown machine. Thus, the task for the learning system is similar to that of machine identification [21][35][32].

By using homing sequences together with experiments, Rivest [53] and Schapire [59] provided a method that does not require a means of resetting an unknown machine to an initial state. They used a notion called diversity, that is, equivalence between experiments or tests [59].

Their notion of diversity can be viewed as equivalency between states, that is, two states are equivalent if every test produces the same value for both states. Their experiments can then be viewed as a distinguishing experiment described in Kohavi [32]. Their system was later implemented by Mozer [44] using neural networks.

3. OVERVIEW OF THE SYSTEM FOR LEARNING SEQUENTIAL TASKS

3.1. Introduction

This chapter provides an overview of the system developed in this investigation. It first shows how to represent internal models and training data. Then it outlines the learning algorithms used in the system.

3.2. Internal models

The learning system uses internal models to record what it has learned. In this investigation, finite state machines (see Definition 1.4) are used as the internal models. Finite state machines are chosen because they can represent sequential tasks in compact formats and because many techniques are available to manipulate them.

<i>State \ Input</i>	01	test	z
s0	s1, 101	s1, go	s0,s1, Out2
s1	s0, 1?1	s0,s2, run	s-1, *
s2	s1, ???	s-1, OK	s0, stop

Figure 3.2.1. Example of an internal model.

Finite state machines are stored internally by using state tables. An example is shown in Figure 3.2.1. In the example, “01, test, z” are inputs. In the first column of the table, “s0, s1, s2” are present states. There is a list associated with each pair of input and present state. The last item in the list is the output, and the remaining items are possible next states. A state is presented internally by an integer (but it is shown in the example preceded by an “s”).

To be able to store incompletely specified state tables and undetermined next states, a next state is represented by a list of integers. Possible next states are listed, such as, “s0, s2”. An unspecified next state is represented by “s-1” (read as “s minus one”) which can be considered to be a list that contains all the states of a state table. In the example a “s-1” is considered as “s0, s1, s2”.

To be able to store binary numbers, text, and don't cares, an input or an output is represented by a string of characters. A don't care bit within a binary number is represented by " ? " which can either be a 0 or an 1, such as, "1?1" can be either "101" or "111". In the example, an entry shown as " s-1, * " indicates an unspecified next state and an unspecified output. An unspecified output, " * ", can be assigned the value of any other output.

3.3. Training data

During training the system receives sequences of input/output examples (see Definition 1.3). An example is shown in Figure 3.3.1. A sequence of input/output examples consists of pairs of inputs and their corresponding outputs. The pairs of input/output examples are ordered in time. Each pair is a step in a sequential task or a combinational task.

<i>Step</i>	0	1	2	3	4	5	6	7
<i>Input</i>	up	up	up	up	up	up	down	down
<i>output</i>	1	2	3	0	1	2	1	0

Figure 3.3.1. A sequence of input/output examples.

Main process

```
FOR each input/output sequence
  IF no internal model exists
    THEN (1) create a new internal model by
      transforming input/output sequence into minimum finite state table
  ELSE FOR each existing internal model
    (2) Check consistency and update internal model by
      comparing input/output sequence to the state table, and by
      incorporating new information into updated state table.
    (3) Handle inconsistent information by
      weighting internal models, and by
      constructing additional models.
```

Figure 3.4.1. Overview of the learning process.

3.4. Learning algorithms

Learning algorithms manipulate training data and organize internal models. An overview of the learning process developed in this investigation is provided in Figure 3.4.1. Each of the three major functions of the learning system is highlighted in the figure.

The process (1) for creating a new internal model is described in detail in Chapter 4. The process (2) for checking consistency and updating

internal models is described in Chapter 5. The process (3) for handling inconsistent information is described in Chapter 6.

Several training examples and their simulation results are provided in Chapter 7. The performance of the system is discussed in Chapter 8.

4. CREATING A NEW INTERNAL MODEL

4.1. Introduction

The training data presented to the system is in the form of a sequence of input/output examples. The system uses a finite state machine to record the learned information. Thus, a process to infer a finite state machine from a sequence of input/output examples must be developed. This chapter describes the process.

4.2. Process for creating a new internal model

The process for creating a new internal model transforms a sequence of input/output examples to a minimal finite state machine.

The minimal finite state machine serves two essential purposes for the learning system.

Process for creating a new internal model

1. Converting sequence of input/output examples
2. Checking compatible
3. Finding maximum compatibility classes
4. Deriving all prime compatibility classes
5. Finding minimum closed cover
6. Deriving a minimum finite state machine

Figure 4.2.1. Steps for creating a new internal model.

First, it records in a compact way the given sequence of input/output examples. Thus when the system sees the same input sequence, it will be able to reproduce the same output sequence.

Secondly, the minimal finite state machine represents an infinite sequence that is the result of generalizing the given finite sequence of examples. Thus when the system sees an input sequence that is similar to but not exactly equal to the given input sequence, it will be able to produce an output sequence that is based on the result of the generalization.

An outline of the process is provided in Figure 4.2.1. The first step is to transform a sequence of input/output examples to an incompletely

specified state table. Then states in the incompletely specified state table are partitioned into compatible classes. Then some of the compatible classes called prime compatible classes are used in a search procedure to find a minimal set of compatible classes that is then transformed into a minimal finite state machine.

Each step outlined in Figure 4.2.1 is described in one of the following sections.

4.2.1. Converting sequence of input/output examples

This section describes the process for converting a sequence of input/output examples to an incompletely specified state table. An example is first used to illustrate the process. Then a more precise description of the process is provided.

Step	Input	Output
0	1	0
1	1	1
2	1	0
3	0	0
4	1	1
5	0	0
6	1	0

Figure 4.2.2. Input/output sequence.

As an example, we will construct an incompletely specified state table from the input/output sequence shown in Figure 4.2.2. In the figure, the input sequence is $\{1,1,1,0,1,0,1\}$ as shown in the input column, and the output sequence is $\{0,1,0,0,1,0,0\}$ as shown in the output column.

The input sequence has two distinct inputs, 1 and 0, so the set of input symbols, $I = \{1, 0\}$, and so there are two input columns in the resultant incompletely specified state table shown in Figure 4.2.3 (Breeding [13]). The output sequence has two distinct outputs, 1 and 0, so the set of output symbols, $O = \{1, 0\}$.

The input/output sequence has seven input/output pairs, so there are seven states for the set of states, S ; each state in S associates with one

Present State	Input	
	1	0
0	1, 0	-1, *
1	2, 1	-1, *
2	3, 0	-1, *
3	-1, *	4, 0
4	5, 1	-1, *
5	-1, *	6, 0
6	-1, 0	-1, *

NextState,Output

Figure 4.2.3. Incompletely specified state table

input/output pair, i.e., $S = \{s_0, s_1, \dots, s_6\}$. The initial state s_0 associates with the first input/output pair, (1, 0). The state s_1 associates with the second pair, (1, 1), and so on.

In Figure 4.2.3, a state is denoted by 0, 1, ..., or 6; a “don’t care” next state is denoted by “-1”; and a “don’t care” output is denoted by “* ”.

The state transition function δ and the output function λ are specified in the figure. For instance, as shown in present state 0, the function $(\delta : I \times S \rightarrow \rho(S))$ maps input 1 and present state 0 to next state 1. It maps input 0 and present state 0 to next state “don’t care”. The output

function $(\lambda : I \times S \rightarrow O)$ maps input 1 and present state 0 to output 0; input 0 and present state 0 to output “don’t care”.

The following describes more precisely the process to construct a finite state machine $(I, O, S, s_0, \delta, \lambda)$, that is represented by an incompletely specified state table, from the information provided by an arbitrary finite input/output sequence $((x_0, y_0), (x_1, y_1), \dots, (x_n, y_n))$.

First, let the set of input symbols, I , equal the distinct inputs in $\{x_0, x_1, \dots, x_n\}$, that is,

$$I = \{x_0\} \cup \{x_1\} \cup \dots \cup \{x_n\}$$

where, the operation $\cup$ is set union. Similarly, let O equal the distinct outputs:

$$O = \{y_0\} \cup \{y_1\} \cup \dots \cup \{y_n\}$$

Next, let one state be associated with one input/output pair; s_k associates with (x_k, y_k) for $0 \leq k \leq n$. Thus, the initial state, s_0 , associates with the first input/output pair (x_0, y_0) , and

$$S = \{s_0, s_1, \dots, s_n\}$$

The state transition function δ maps an input i in I and state s_k to state s_{k+1} if $i = x_k$ and $0 \leq k \leq n-1$; otherwise, it is not specified and is considered as “don’t care”. That is,

$$\delta(i, s_k) = \begin{cases} s_{k+1}, & \text{if } i = x_k \text{ and } 0 \leq k \leq n-1 \\ \text{don't care}, & \text{if } i \neq x_k \text{ and } 0 \leq k \leq n-1 \\ \text{don't care}, & \forall i \in I \text{ and } k = n \end{cases}$$

The output function λ maps an input i in I , and state s_k to an output $o = y_k$ if $i = x_k$ and $0 \leq k \leq n$; otherwise, it is not specified and is considered as “don’t care”. That is,

$$\lambda(i, s_k) = \begin{cases} y_k, & \text{if } i = x_k \text{ and } 0 \leq k \leq n \\ \text{don't care}, & \text{otherwise.} \end{cases}$$

The resultant incompletely specified state table defines a machine that is consistent with the given sequence of input/output examples. The following sections will show a process for the generalization that uses a special process for minimizing the incompletely specified state table.

4.2.2. Checking compatible

To generalize and to simplify the incompletely specified state table defined in the last section, we will first compare any two states to determine whether they are compatible [32].

Definition 4.1. Let $o_{ij} = \lambda(i, s_j)$, and $s_{ij} = \delta(i, s_j)$. State s_j is *compatible* to state s_k (denoted by $s_j \sim s_k$) if for all input i in I , each output o_{ij} equals o_{ik} , and each next state s_{ij} is compatible to s_{ik} , that is,

$$s_j \sim s_k \text{ if } \forall i \in I \ni o_{ij} = o_{ik} \wedge s_{ij} \sim s_{ik}$$

An output o_{ij} equals an output o_{ik} if they are identical in symbol, or if one of the outputs is “don’t care”. If each output is coded with several characters, then two outputs are equal if each of their corresponding characters are equal.

A next state s_{ij} is *compatible* to a next state s_{ik} if either one of them is a “don’t care” or both are “don’t cares”. If neither of them is a “don’t care”, then further checking is needed to determine whether s_{ij} and s_{ik} are compatible. The outcome in turn determines whether s_j and s_k are

compatible. On the other hand, if s_j and s_k are compatible, then s_{ij} and s_{ik} must be compatible. That is,

$$s_j \sim s_k \text{ imply } s_{ij} \sim s_{ik}$$

For a machine defined by an incompletely specified state table that is generated from an input/output sequence, the next state s_{ij} can only be s_{j+1} or “don’t care”, and the next state s_{ik} can only be s_{k+1} or “don’t care”. Thus,

$$s_j \sim s_k \text{ can only imply } s_{j+1} \sim s_{k+1}$$

Based on the above property, a special method for checking compatibility is developed in this investigation. As an example, we will determine whether any two states shown in Figure 4.2.3 are compatible. The results are recorded in Figure 4.2.4. In the figure, “0 1 --> x”, for example, denotes that state 0 and state 1 are not compatible. “0 2 --> ” denotes that for state 0 and state 2 to be compatible, state 1 and state 3 must be compatible, that is, $s_0 \sim s_2$

0 1 --> x	1 2 --> x	2 3 --> ~	3 4 --> ~	4 5 --> ~	5 6 --> ~
0 2 -->	1 3 --> ~	2 4 --> x	3 5 --> (x)	4 6 --> x	
0 3 --> ~	1 4 -->	2 5 --> ~	3 6 --> ~		
0 4 --> x	1 5 --> ~	2 6 --> ~			
0 5 --> ~	1 6 --> x				
0 6 --> ~					

States m n imply --> m+1 n+1; compatible ~; or not compatible x.

Figure 4.2.4. A modified implication table.

implies $s_1 \sim s_3$. The “0 3 --> ~” denotes that state 0 and state 3 are compatible.

The method for checking compatibility consists of two stages. First, the method checks each pair of states, in order {0 1}, {0 2}, ..., {0 6}, {1 2}, {1 3}, ..., {5 6}, to determine whether it is compatible, or it implies another pair of states. For example, states {0 1} are not compatible because as shown in row 0 and row 1 of Figure 4.2.3, state 0 and state 1 given the same input 1 have different outputs. As another example, states {3 5} imply states {4 6}, because for all inputs, state 3 and state 5 have equal outputs; also for states {3 5} to be compatible, states {4 6} must be compatible.

Second, the method checks each pair of states row by row and from right to left. If states $\{m\ n\}$ are not compatible but the states $\{m-1\ n-1\}$ imply the former, then states $\{m-1\ n-1\}$ are marked with an x. For example, as shown in Figure 4.2.4, states $\{4\ 6\}$ are not compatible. As a result of the first stage, states $\{3\ 5\}$ implies states $\{4\ 6\}$. Now, as a result of this stage, states $\{3\ 5\}$ are marked with an x.

4.2.3. *Finding maximum compatibility classes*

In the last section we considered whether two states are compatible while in this section we will consider whether a set having three or more states is compatible. We will define a set called maximum compatibility class, and will derive a procedure to find all the maximum compatibility classes.

Definition 4.2. The relation that state s_j is compatible to s_k is called a compatibility relation. A subset of set S having all its states compatible to each other is called *compatibility class*. Since compatibility relation is not transitive, i.e., $s_h \sim s_j$ and $s_j \sim s_k$ do not imply $s_h \sim s_k$, a

Figure 4.2.5. Finding maximum compatibility classes.

compatibility class must have all its states pair-wise compatible. A *maximum compatibility class* is a compatibility class that is not a subset of any other compatibility class [32].

The following example shows a procedure developed in this investigation for deriving maximum compatibility classes from the set S. The example is based on the compatible information provided in Figure 4.2.4. The procedure produces a binary search tree that is shown in Figure 4.2.5.

The procedure starts with the set S. It checks any two states in S. If two states m and n are not compatible, then it splits S into two sets: one set contains all states except n, and the other contains all states

except m. Next, it checks the two sets, one at a time using the same method.

In our example, $S = \{0\ 1\ 2\ 3\ 4\ 5\ 6\}$, and states 0 and 1 are not compatible (see Figure 4.2.4), so S is split into $\{0\ 2\ 3\ 4\ 5\ 6\}$ and $\{1\ 2\ 3\ 4\ 5\ 6\}$. As shown in Figure 4.2.5, two states that are not compatible are underlined. Next, the set $\{0\ 2\ 3\ 4\ 5\ 6\}$ is checked and split into two sets. Then, the set $\{1\ 2\ 3\ 4\ 5\ 6\}$ is checked and split. The procedure continues.

If a set has already been checked, then it is not checked again; those sets are shown in *italic* in Figure 4.2.5. If a set is checked but not split, then it is a compatibility class. Maximum compatibility classes are shown in **bold**. In this example, the set $\{2\ 3\ 6\}$ is a compatibility class, but it is not a maximum compatibility class because it is a subset of the compatibility class $\{0\ 2\ 3\ 6\}$.

4.2.4. *Deriving all prime compatibility classes*

From Definition 4.2 provided in the last section, any subset of a maximum compatibility classes is a compatibility class. Of all those compatibility classes, we will show in the next section that only some of those classes called “prime compatibility classes” are needed for our purpose to derive a minimum finite state machine. In this section, we will define “prime compatibility class”, and will provide a procedure to find all prime compatibility classes.

Definition 4.3. A compatibility class C_p *excludes* a compatibility class C_e if C_p is a proper superset of C_e , and the class D_p implied by C_p is a subset of the class D_e implied by C_e . That is,

$$C_p \text{ excludes } C_e \text{ if } C_p \supset C_e \text{ and } D_p \subseteq D_e$$

where, C_p implies D_p , and C_e implies D_e . The class C_e is said to be excluded by C_p [24].

Max. Comp.		Implied Class
{0 2 3 6}	-->	{1 3}
{0 2 5 6}	-->	{1 3}
{1 3 4}	-->	{2 5}
{1 4 5}	-->	{2 5}

Figure 4.2.6. Maximum compatibility classes and their implied classes.

Definition 4.4. A *prime* compatibility class is a compatibility class that is not excluded by any other compatibility classes [24].

All maximum compatibility classes are prime. A subset of a maximum compatibility class is a compatibility class. To find all prime compatibility classes, a compatibility class is compared with all existing prime compatibility classes; if it is not excluded, then it is a prime.

The following example shows a procedure to derive all prime compatibility classes. The procedure is identical to that reported in Grasselli [24] but is provided here for completeness.

The procedure starts with maximum compatibility classes. Using the compatible information provided in Figure 4.2.4, the procedure first

produces an implied class for each of the maximum compatibility classes. The results are shown in Figure 4.2.6.

As an example, consider maximum compatibility class $\{0\ 2\ 3\ 6\}$. The procedure checks each pair of states in the class: states $\{0\ 2\}$ imply states $\{1\ 3\}$; states $\{0\ 3\}$, $\{0\ 6\}$, $\{2\ 3\}$, $\{2\ 6\}$, and $\{3\ 6\}$ are compatible pairs (see Figure 4.2.4). Thus, for the set $\{0\ 2\ 3\ 6\}$ to be a compatibility class implies that the set $\{1\ 3\}$ must be a compatibility class.

Next, the procedure includes as a prime class the maximum compatibility class that contains the most states. Let m be the number of states in the class. From the class, the procedure generates subsets that contain $m-1$ states. It compares each of the subsets with all existing prime classes; if a subset is not excluded by any existing prime classes, then it is included as a prime class.

The procedure next includes maximum compatibility classes that contain $m-1$ states. It then generates and compares subsets that contain

Prime Comp.		Implied Class
{0 2 3 6}	-->	{1 3}
{0 2 5 6}	-->	{1 3}
{2 3 6}	-->	$\emptyset$
{0 3 6}	-->	$\emptyset$
{2 5 6}	-->	$\emptyset$
{0 5 6}	-->	$\emptyset$
{1 3 4}	-->	{2 5}
{1 4 5}	-->	{2 5}
{3 4}	-->	$\emptyset$
{1 3}	-->	$\emptyset$
{4 5}	-->	$\emptyset$
{1 5}	-->	$\emptyset$

Figure 4.2.7. All prime compatibility classes.

m-2 states. The process continues until subsets that contain one state is generated and compared.

In our example, two maximum compatibility classes contain the most states: they are {0, 2, 3, 6} and {0, 2, 5, 6}; and $m = 4$. They are included as prime compatibility classes as shown in first two rows in Figure 4.2.7. Then, subsets that contain 3 states are generated and compared. The class {0 2 3 6}, for example, has the 3-state subsets and their implied classes as: {0 2 3} --> {1 3}, {0 2 6} --> {1 3}, {2 3 6} --> $\emptyset$, and {0 3 6} --> $\emptyset$; where “ --> $\emptyset$ ” means that the class is compatible in itself. Of those four subsets, the classes {0 2 3}

and $\{0\ 2\ 6\}$ are excluded by $\{0\ 2\ 3\ 6\}$; and the classes $\{2\ 3\ 6\}$ and $\{0\ 3\ 6\}$ are prime. The 3-state subsets of the class $\{0\ 2\ 5\ 6\}$ are then generated and compared.

Next, maximum compatibility classes $\{1\ 3\ 4\}$ and $\{1\ 4\ 5\}$ are included as shown in bold in Figure 4.2.7. Then, 2-states subsets of each of the 3-state prime compatibility classes are generated and compared. Finally, 1-state subsets of each of the 2-state prime compatibility classes are generated and compared. The resultant prime compatibility classes are shown in Figure 4.2.7.

4.2.5. Finding a minimum closed cover

To derive a minimum finite state machine, we will use a concept called “minimum closed cover”. We will show in this section that we can find a minimum closed cover in the set of all prime compatibility classes that was derived in the last section.

Definition 4.5. A set C containing compatibility classes is *closed* if for each compatibility class C_k in C such that C_k implies D_k , and D_k is a subset of at least one compatibility class in C . That is,

$$C \text{ is closed if } \forall C_k \in C, \exists x \in C \ni C_k \text{ implies } D_k, \text{ and } D_k \subseteq x$$

Definition 4.6. A compatibility class C_k *covers* a state s if s is in C_k .

A set of compatibility classes covers a machine if all states of the machine are covered. A set of compatibility classes is called *closed cover* if it is closed and it covers a machine. A *minimum closed cover* of a machine is a closed cover that has the minimum number of compatibility classes [32].

Theorem: The set of all prime compatibility classes contains a minimum closed cover [24].

To find a minimum closed cover, a breadth-first search procedure is used. The procedure described here is based on that proposed by Meisel [41] that is modified here to become a breadth-first search

Figure 4.2.8. Search tree for finding minimum closed cover.

procedure. As the search progresses, a tree is generated (see an example shown in Figure 4.2.8).

The search begins at the root node. The root node generates its child-nodes, and checks for closed and cover conditions. Then, each of its child-nodes generates their child-nodes, and checks for closed and cover conditions. The process continues until both closed and cover conditions are satisfied.

For our search tree, the root node is called level 0; its child-nodes are called level 1; its grandchild-nodes are called level 2, and so on. As shown in Figure 4.2.8, each node contains three items: a prime

compatibility class (the top list), a list called “pending list” (the middle list), and a list called “missing list” (the bottom list).

There is a unique path from the root node to any other node. A path from the root node to a node in level 2, for example, connects the root node, a node in level 1, and the node in level 2. There is a prime compatibility class in each of the connected nodes. The collection of all those prime compatibility classes, except the one in the root node, is called a *path set*. The path set of a node in level 2, for example, contains 2 compatibility classes. In general, the path set of a node in level n contains n compatibility classes. For example, in Figure 4.2.8 the path set for the left-most node in level 2 contains the compatibility classes $\{0\ 2\ 3\ 6\}$ and $\{1\ 3\ 4\}$.

For any node, its pending list determines whether its path set is closed. If the pending list is empty, then the path set is closed. If it is not empty, then it contains implied classes that have not yet been closed.

Also for any node, its missing list determines whether its path set covers all states. If the missing list is empty, then the path set covers all

states. If it is not empty, it contains states that have not yet been covered.

If a node has both an empty pending list and an empty missing list, then its path set is a closed cover. In Figure 4.2.8, the right-most node in level 2 is an example.

How a node generates its child-nodes depends on whether its pending list is empty. If its pending list is not empty, then it takes an implied class out of the pending list, and generates a node for each prime compatibility class that contains the implied class. If its pending list is empty, then it finds from its missing list a state that is covered by a minimum number of prime compatibility classes, and generates a node for each prime compatibility class that covers the state.

A new child-node contains a different prime compatibility class than its parent-node. It inherits the pending list and the missing list, but updates them if needed. If its prime compatibility class has implied classes, then it adds those implied classes to its pending list. But, it removes

from its missing list, the states that are covered by its prime compatibility class.

For our example, we will find a minimum closed cover from the set of all prime compatibility classes shown in Figure 4.2.7. The resultant search tree is shown in Figure 4.2.8.

First, we let the root node contain an empty prime compatibility class, an empty pending list, and a missing list that contains all states. Since the root node has an empty pending list, it finds from its missing list a state that is covered by a minimum number of prime compatibility classes. States 0, 1, 2, and 4 are each covered by four prime compatibility classes; states 3, 5, and 6 are each covered by six classes. Among the former, state 0 is arbitrarily chosen and it is covered by $\{0\ 2\ 3\ 6\}$, $\{0\ 2\ 5\ 6\}$, $\{0\ 3\ 6\}$, and $\{0\ 5\ 6\}$. Four nodes are generated and their pending list and missing list are updated as shown in level 1 in Figure 4.2.8. The class $\{0\ 2\ 3\ 6\}$, for example, implies $\{1\ 3\}$ which is put in the pending list; the states that have not yet been covered are $\{1\ 4\ 5\}$, which are put in the missing list.

Next, the left-most node in level 1 generates its child-nodes. Since its pending list contains $\{1\ 3\}$, it removes $\{1\ 3\}$ from its pending list, and generates one node for each prime compatibility classes that cover $\{1\ 3\}$, as shown in level 2 in Figure 4.2.8.

One node is generated at a time, and closed and cover conditions are checked. Because the conditions are not yet satisfied, the search continues. Now, the next node from the left in level 1 generates its child-node. This time, the generated node has both an empty pending list and an empty missing list. Thus, a minimum closed cover has been found and it is $\{0\ 2\ 5\ 6\}$ and $\{1\ 3\ 4\}$.

4.2.6. Deriving a minimum finite state machine

This section describes a process for deriving a minimum finite state machine $M' = (I', O', S', s_0', \delta', \lambda')$ from a minimum closed cover $\mu = \{C_0, C_1, \dots, C_m\}$, and the original finite state machine $M = (I, O, S, s_0, \delta, \lambda)$. The process is developed in this investigation.

Present State	Input	
	1	0
0	1, 0	-1, *
1	2, 1	-1, *
2	3, 0	-1, *
3	-1, *	4, 0
4	5, 1	-1, *
5	-1, *	6, 0
6	-1, 0	-1, *

NextState, Output

(a) Original state table.

Present State	Input	
	1	0
A : { 0 2 5 6 }	B, 0	A, 0
B : { 1 3 4 }	A, 1	B, 0

NextState, Output

(b) Minimum state table.

Figure 4.2.9. Result of state reduction.

An example is first used to illustrate the process. Then a more precise description is given.

As an example, we will derive a minimum finite state machine from the minimum closed cover found in the last section and from the original finite state machine. The state tables of the original finite state machine and of the resultant minimum finite state machine are shown in Figure 4.2.9. In this example, the set of minimum closed cover $\mu = \{\{0\ 2\ 5\ 6\}, \{1\ 3\ 4\}\}$, the set $I' = I = \{1, 0\}$, and the set $S' = \{s_0', s_1'\}$. In Figure 4.2.9(b), the state A represents s_0' which is assigned to the

compatibility class $\{0\ 2\ 5\ 6\}$, and the state B represents s_1' which is assigned to the compatibility class $\{1\ 3\ 4\}$.

The next state for state A given input 1, for example, is specified because there exists a state in $\{0\ 2\ 5\ 6\}$ of that state the next state is specified. In this case, the next state for state 0 is state 1, and that for state 2 is state 3 (see Figure 4.2.9(a)). Both state 1 and state 3 are contained in $\{1\ 3\ 4\}$, so the next state for state A given input 1 is state B. The next state for state A given input 0 is state A because state 5 has next state 6 which is contained in $\{0\ 2\ 5\ 6\}$.

The output for state A given input 1, for example, is specified because there exists a state in $\{0\ 2\ 5\ 6\}$ of that state the output is specified. In this case, the outputs of states 0, 2, and 6 are each equal 0, so the output for state A given input 1 is 0. To complete our example, the set O' is equal to $\{0, 1\}$.

The resultant minimum state table (Figure 4.2.9(b)) represents the required minimum finite state machine that is a newly created internal model.

The following describes more precisely the process for deriving a minimum finite state machine $M' = (I', O', S', s_0', \delta', \lambda')$ from a minimum closed cover $\mu = \{C_0, C_1, \dots, C_m\}$, and the original finite state machine $M = (I, O, S, s_0, \delta, \lambda)$.

First, let the set of inputs I' of the minimum finite state machine equal I of the original finite state machine. Next, let the set of states $S' = \{s_0', s_1', \dots, s_m'\}$. Each state in S' is assigned to one compatibility class in μ . The initial state s_0' is assigned to a compatibility class in μ such that s_0 is in that compatibility class. Without loss of generality, let that compatibility class be C_0 . That is, s_0' is assigned to C_0 and $s_0 \in C_0$. Assign each of the remaining states s_j' to the corresponding compatibility class C_j , for $1 \leq j \leq m$.

For an input $i_k' \in I'$ and a state $s_j' \in S'$, the next state $\delta'(i_k', s_j')$ is specified if there exists a state $s_h \in C_j$ such that $\delta(i_k, s_h)$ is specified; otherwise the next state is not specified. If $\delta'(i_k', s_j')$ is specified, it is equal to state $s_x' \in S'$ such that $\delta(i_k, s_h)$ is in C_x ; otherwise, it is equal to “don't care”. That is,

$$\delta'(i_k', s_j') = \begin{cases} s_x', & \text{if } \exists s_h \in C_j \ni \delta(i_k, s_h) \in C_x \\ \text{don't care,} & \text{otherwise.} \end{cases}$$

Similarly, the output $\lambda'(i_k', s_j')$ is specified if there exists a state $s_h \in C_j$ such that $\lambda(i_k, s_h)$ is specified; otherwise the output is not specified. If it is specified, it is equal to $\lambda(i_k, s_h)$; otherwise, it is equal to “don't care”. That is,

$$\lambda'(i_k', s_j') = \begin{cases} \lambda(i_k, s_h) & \text{if } \exists s_h \in C_j \ni \lambda(i_k, s_h) \text{ is specified} \\ \text{don't care,} & \text{otherwise.} \end{cases}$$

This definition is used for outputs each of which is encoded by only one character or one bit. If each output is encoded by more than one

character or one bit, each character or each bit is defined separately in the same way as the one character case.

The definition of the output function λ' specifies how each output o_{kj}' should be, for each input $i_k' \in I'$ and each state $s_j' \in S'$. The set of outputs O' is equal to set union of all set $\{o_{kj}'\}$, for all input $i_k' \in I'$ and all state $s_j' \in S'$.

The process for creating a new internal model is completed. It transforms a sequence of input/output examples to a minimum finite state machine. The resultant minimum finite state machine is the new internal model.

5. CHECKING CONSISTENCY AND UPDATING INTERNAL MODELS

5.1. Introduction

The last chapter describes the process for creating a new internal model. This chapter describes how to compare a sequence of input/output examples with an existing internal model, and describes how to create an updated internal model to incorporate new information.

5.2. Process for checking consistency and updating internal model

The process outlined in Figure 5.2.1 is developed in this investigation for both checking consistency and updating internal model. As highlighted in the figure the process incorporates new information into an internal model by adding new input columns, specifying don't cares, and removing possible next states.

Checking consistency and updating internal model

Compare the given input sequence to the state table, starting at s_0

IF an input is not in the state table

THEN **add new input column** in the state table.

In responding to the given input sequence,
the state table generates many output sequences.

FOR each output sequence generated by the state table

Compare it with the given output sequence and **specified don't cares**

IF they are *compatible*

THEN keep the generated output sequence and keep the specified don't cares

ELSE remove the generated output sequence by

removing a possible next state

that is responsible for generating the output sequence,
and undo the specified don't cares.

IF the updated state table contains an entry that
has all its possible next state removed

THEN both the updated state table and the original state table
are not *consistent* with the input/output sequence

ELSE either the original or the updated state table is *consistent* with
the input/output sequence.

Figure 5.2.1. Checking consistency and updating internal model.

An example is first used to illustrate some functions of the process.

Details are then provided in the following sections.

As an example, an input/output sequence (0/11, n/01, 1/?1) is compared to the state table shown in Figure 5.2.2a. The result of each step of the comparisons is shown in Figure 5.2.2b while the resulting

<i>State \ Input</i>	0	1
s0	s1, 1?	s1, 00
s1	s0, 10	s0, 01

a. Original state table.

Given Inputs	Produced (a)	Produced (b)	Required Outputs
0	s0, 1?	=	11
n	s1, 01	=	01
1	s0, 00	s1, 01	?1

b. Sequences produced after adding new input column.

<i>State \ Input</i>	0	1	n
s0	s1, 11	s1, 00	s-1, *
s1	s0, 10	s0, 01	s1, 01

c. Resulting state table.

Figure 5.2.2. Example for updating internal model.

modified state table is shown in Figure 5.2.2c. The steps of the comparison are as follows.

(1) Starting at state s0, the state table receives an input 0. It produces an output 1? which is compared with the required output 11 (shown in Figure 5.2.2b). To make the outputs compatible, the produced output

1? is now specified to 11. Since the next state is s1, the state table goes to state s1.

(2) In state s1, the state table receives an input n which is not already in the state table. The state table creates a new input column that is labeled n (shown in Figure 5.2.2c). All the entries in the new column are initialized to don't cares: don't care next state and don't care output; shown as "s-1, *" in the table (Figure 5.2.2c shows the final resultant state table not the result of this step). The entry at the current state that is state s1 is then set to "s-1, 01" in which the next state is a don't care while the value of the output is that of the input/output pair "n/ 01" because at the current state the state table is required to produce an output 01. The don't care next state "s-1", at the current state s1 and input n, is treated as next states s0 and s1 that are all possible states in the table. Thus, there are two possible next states for the current state. The next state s0 is checked first while next state s1 is saved in a stack to be checked later. Now the state table goes to state s0.

(3a) In state s_0 , the state table receives an input 1. It then produces an output 00 which is compared to required output ?1. The first bit can be compatible if the don't care bit is set to 0, but the second bit can not be compatible. So the required output is restored back to ?1. The result is that these two outputs are not compatible. This completes the sequence (a) as shown in Figure 5.2.2b. This sequence is not compatible to the required output sequence, because the output produced in this step is not compatible to the corresponding required output. Thus, the sequence needs to be removed. To do so, the possible next state s_0 that is the result of step (2) is removed.

(3b) To generate sequence (b) (see Figure 5.2.2b), the process performs back-tracking and finds that in step (2) there is another possible next state needed to be checked. It retrieves from the stack the possible next state that is state s_1 . The state table now goes to state s_1 in which it receives input 1 and then produces output 01 which is compared to the required output ?1. These two outputs are compatible and the required output is now specified to 01. This complete sequence (b). In the figure, the “ = ” sign in the entry indicates that the value of

that entry is equal to that on the left. To complete sequence (b) the process does not need to regenerate those values that are equal to that already generated in the sequence (a). Sequence (b) is compatible with the required output sequence. Thus, the possible next state s_1 is kept. This completes the entire search process and the final result is shown in Figure 5.2.2c.

5.3. Adding new input columns

A new input column is added to a state table whenever a sequence of input/output examples contains a new input symbol that is not already in the state table. Adding new input columns allows the learning system to expand the scope of its internal models. The new information helps the learning system to generalize its internal models or to make its internal models cover more instances. The process for adding a new input column is illustrated in step (2) in the last example.

5.4. Checking consistency

For the learning system to decide what to do with the new information, it must have a method to compare the new information with its existing internal models. The new information comes to the learning system in the form of sequences of input/output examples, while the system stores its internal models in the form of state tables. Thus a method for comparing a sequence of input/output examples with a state table must be developed.

This investigation develops a method for comparing a sequence of input/output examples with a state table that can be completely specified or incompletely specified. A completely specified state table can produce only one output sequence in responding to one input sequence. An incompletely specified state table, on the other hand, can produce many output sequences in responding to one input sequence. For example, after a new input column is added to the state table shown in Figure 5.2.2a, it can produce two output sequences ((a) and (b) in Figure 5.2.2b) in responding to the given input sequence.

Definition 5.1. A completely or incompletely specified state table is *consistent* with a given input/output sequence if in responding to the given input sequence, the state table produces at least one output sequence that is compatible with the given output sequence, and produces no output sequence that is not compatible with the given output sequence.

Definition 5.2. An output sequence $(p_1, p_2, p_3, \dots, p_m)$ is *compatible* with another output sequence $(q_1, q_2, q_3, \dots, q_n)$ if m equals n , and output p_i is compatible with output q_i , for $1 \leq i \leq m$.

Definition 5.3. An output $c_1c_2c_3\dots c_m$ is *compatible* with another output $d_1d_2d_3\dots d_n$ if m equals n , and character c_i is identical to character d_i , or either c_i or d_i is a don't care character, for $1 \leq i \leq m$. A don't care character is *specified* to a non-don't care character, if it is compared to the non-don't care character. A don't care output is a special output that is compatible with any other output. Once a don't care output is compared to a non-don't output, it is specified to the non-don't output.


```

Sequence O (o1, o2, o3, o4)
      O = (a,  c,  b,  c )
Initial value: a = 11?  b = ?10   and   c = *

Sequence P (p1, p2, p3, p4)
      P = (x,  y,  x,  y )
Initial value: x = ?10   and   y = 1?0

Compare o1: a=11? to p1: x=?10   →   a=110   x=110
Compare o2: c= *   to p2: y=1?0   →   c=1?0   y=1?0
Compare o3: b=?10 to p3: x=110   →   b=110   x=110
Compare o4: c=1?0 to p4: y=1?0   →   c=1?0   y=1?0

```

Figure 5.5.1. Example for comparing two sequences.

5.5. Comparing output sequences and specifying don't cares

Comparing two output sequences in which any output may contain don't care characters or may be a don't care output requires a method for comparing two outputs and a method for keeping track of which don't care character or which don't care output has been specified.

An example shown in Figure 5.5.1 will illustrate the method. There are two output sequences say sequence O (o1, o2, o3, o4) and sequence P (p1, p2, p3, p4). To complicate the matter, sequence O makes up only three distinct output variables namely a, b, and c; those values will

change as the result of the comparisons. Sequence O = (a, c, b, c). The initial value of a = 11?, b = ?10, and c = * , where “ * ” is a don’t care output, and “ ? ” is a don’t care character. Sequence P makes up of only two distinct output variables namely x and y. The initial value of x = ?10 and y = 1?0 . Sequence P = (x, y, x, y).

1. Start the comparison with the first pair, o1 to p1. Compare o1 (a=11?) to p1 (x=?10) resulting in that this pair is compatible and that the values of a and x are both changed. Now (a=110) and (x=110) as well. The original don’t care character in a is now specified to 0 while that of x is specified to 1.
2. Next compare o2 (c=*) to p2 (y=1?0) resulting in compatible and the value of c is changed while that of y is unchanged. Now (c=1?0) and (y=1?0); the original don’t care output c is now specified to the specific value.
3. Next compare o3 (b=?10) to p3 (x=110); where the value of x is the result of the first comparison. The result of the comparison is

compatible and the value of b is changed. Now (b=110) and (x=110).

4. Finally compare o4 (c=1?0) to p4 (y=1?0) resulting in compatible and both the value of c and y are unchanged. Since all the corresponding pairs of sequence O and P are compatible, the final result is that these two sequences are compatible.

In this investigation, it is required to compare a given output sequence to many output sequences generated by a state table. For the purpose of creating a modified state table that is consistent with the given input/output sequence, a generated output sequence will be removed if it is not compatible with the given output sequence. The specified don't cares, associated with the removed output sequence, will be restored to don't cares.

As described in the last example (Figure 5.5.1) some of the don't care characters may be specified to a specific value and some don't care outputs may be specified. The results of the specification are kept

Figure 5.6.1. Transform sequences into a search tree.

permanent if the two sequences are compatible, and are undone if the two sequences are not compatible.

5.6. Searching and backtracking

Checking whether a state table is consistent with a given sequence of input/output examples requires the comparison of the given output sequence with the output sequences generated by the state table. Since there are many cases that not all the output sequences are required to be generated or compared, a depth-first search method is used.

An example (shown in Figure 5.6.1) will illustrate how a set of output sequences can be viewed as a search tree. In the search tree, each path from the root node to a leaf node represents an output sequence. The number of output sequences equals the number of leaf nodes in the

search tree. The number of outputs in the sequences equals the number of levels in the search tree. The first output in the output sequence corresponds to the first level in the search tree, the second to the second and so on.

Comparing one whole output sequence is equivalent to traversing the search tree from the root node to a leaf node. Comparing all the output sequences one at a time is equivalent to traversing the search tree from the root node to a leaf node and then to another leaf node one at a time. To traverse the search tree in a depth-first method, a subroutine will generate and compare nodes in the order (1, 2, 3, 4, 5, 6) as shown in the search tree in Figure 5.6.1. The subroutine generates and compares one node at a time starting from the root node and continuing to the left-most leaf node. Once it reaches a leaf node or as soon as any node is not compatible, it goes back-track to the most recent cross road and starts searching another branch.

5.7. Removing possible next states

Removing the next state corresponding to an output sequence will remove the output sequence. In order to produce a modified state table that can be consistent with a given input/output sequence, the output sequence generated by a state table will be removed if it is not compatible with the given output sequence. Removing next states allows the learning system to restrict the scope of its internal models. The new information helps the learning system to distinguish possible outcomes, to limit the choices of many outcomes, or to eliminate non-deterministic states.

State \ Input	0	1
s0	s0,s1, 1	s1, 0
s1	s0, 0	s0, 1

a. State table.

Given Inputs	Produced (a)	Produced (b)	Given Outputs
0	s0, 1	s0, 1	1
1	s0, 0	s1, 1	0
1	s1, 1	s0, 0	1

b. Sequences produced.

Figure 5.7.1. Example for removing possible next states.

As an example a given input/output sequence (0/1, 1/0, 1/1) will be compared to the state table shown in Figure 5.7.1a. The result of the comparison is outlined in Figure 5.7.1b. Using state s0 as the starting state, the state table goes through two state sequences and produces two output sequences in responding to the given input sequence. The reason for the two sequences is that there are two possible next states for the present state s0 and input 0. In the two possible next states, s0 is responsible for producing output sequence (a) while s1 is responsible for producing output sequence (b). Compare the required output

sequence to output sequences (a) and (b) resulting in that sequence (a) is compatible and sequence (b) is not compatible. Since the possible next state s_1 is responsible for producing output sequence (b), removing the s_1 will result in a modified state table that will not produce output sequence (b) in responding to the given input sequence. After removing sequence (b), the modified state table is now consistent with the given input/output sequence while the original state table is not.

The process is more complicated if more than one possible next states are responsible for producing an output sequence. Then, there are choices as to which possible next state should be removed in order to remove the output sequence. The result of choosing one possible next state over another may effect whether a modified state table is consistent with a given input/output sequence.

State \ Input	x	y	z
s0	s0,s1, L	s1, L	s0, M
s1	s0, K	s0,s2, L	s1, N
s2	s1, M	s0, K	s0, M

a. State table.

Inputs	(a)	(b)	(c)	Outputs
x	s0, L	s0, L	s0, L	L
y	s0, L	s1, L	s1, L	L
z	s1, N	s0, M	s2, M	N

b. Sequences produced.

Figure 5.7.2. Choosing possible states to be removed.

As an example the input/output sequence (x/L, y/L, z/N) is compared to the state table shown in Figure 5.7.2a. In responding to the input sequence (x, y, z), the state table produces three state sequences and output sequences as shown in Figure 5.7.2b. Compare the required output sequence (L, L, N) to the three produced output sequences resulting in that output sequence (a) is compatible while output sequence (b) and (c) are not compatible. Both sequence (b) and (c) must be removed in order to produce a modified state table that is consistent with the given input/output sequence.

There are two possible next states that are responsible for producing sequence (b): the possible next state s_1 of present state s_0 and input x , and the possible next state s_0 of present state s_1 and input y . Removing any one of the two possible next states will remove the possibility to produce sequence (b). Similarly there are two possible next states that are responsible for producing sequence (c): the s_1 of present state s_0 and input x , and the s_2 of present state s_1 and input y . Removing any one of the two possible next states will remove sequence (c).

To remove both sequence (b) and (c), there are two choices: (1) remove the possible next state s_1 of present state s_0 and input x ; that possible next state is responsible for producing both the sequences, or (2) remove both possible next states s_0 and s_2 of present state s_1 and input y . If choice (1) is chosen, then the resultant modified state table will be consistent with the given input/output sequence and the modified state table will remain consistent with those input/output sequences that are consistent with the original state table. If choice (2) is chosen, then the resultant modified state table will not consistent with those input/output sequences that are consistent with the original

state table. The reason for the later is that the modified state table will now have an undefined next state that of present state s_1 and input y ; in which all possible next states have been removed.

A general rule for choosing which possible next state to remove is:

- (1) remove first those next states that are responsible for as few output sequences as possible; if the result of (1) is that all the possible next states of a particular present state and input have been removed, then
- (2) restore all those possible next states and then remove those next states that are responsible for the whole set of output sequences.

The reason for the general rule is that during the process for creating an updated internal model, it is necessary to ensure that (1) the updated model is consistent with the new input/output sequence and (2) the updated model remains consistent with those old input/output sequences that are consistent with the original state table. To ensure these, the updated state table must have remaining at least one next state for every pair of present states and inputs.

6. HANDLING INCONSISTENT INFORMATION

6.1. Introduction

If a sequence of input/output examples is not *consistent* (see Definition 5.1) with an existing finite state machine then there is inconsistent information. Inconsistent information may come from the environment or from the trainer of the learning system. It may be the result of noise or an error made by the trainer. It may also be the result of examples that describe a task that is different from the task specified by an existing finite state machine.

This chapter describes the process developed in this investigation for handling inconsistent information.

6.2. Process for handling inconsistent information

The process for handling inconsistent information (outlined in Figure 6.2.1) uses other processes described earlier.

- (1) It detects any inconsistency by using the process for checking consistency (described in Section 5.2).
- (2) It incorporates new useful information into the existing models by using the processes for updating internal models (described in Section 5.2).
- (3) It records inconsistent information by using the process of creating a new internal model (described in Section 4.2).

Handling inconsistent information

The weight for a newly created internal model equals 1

FOR each internal model

IF (1) an input/output sequence *is consistent with* the internal model

THEN increase by one the weight of the internal model

ELSE IF the input/output sequence *is consistent with*
an updated internal model

THEN (2) store the updated internal model,

its weight equals one plus that of the original internal model.

IF the input/output sequence *is not consistent with* any existing models

THEN (3) create a new internal model

its weight equals one.

Figure 6.2.1. Process for handling inconsistent information.

A number called “weight” is designed to each internal model. The weight for a newly created internal model equals 1 (as outlined in Figure 6.2.1). Whenever the learning system receives a new sequence of input/output examples, it will check for consistency of the new sequence against all the existing internal models. (Step (1) in Figure 6.2.1) If an internal model is consistent with the new sequence, then its weight is increased by one; otherwise its weight is unchanged. (2) If an internal model can be modified so that the modified model is consistent with the new sequence and that remains consistent with the old sequences, then the newly modified internal model is added and its

weight is one plus the weight of the original internal model. (3)

Otherwise, a completely new internal model is constructed based on the information provided by the new sequence of input/output examples, and its weight is one.

If inconsistent information occurs less frequently than useful information, then the weights of the models that result from inconsistent information will be lower than the weights of the good models. The model having the highest weight is the most favorable model.

7. TRAINING EXAMPLES AND SIMULATION RESULTS

7.1. Introduction

Chapter 3 to Chapter 6 described the operations of the learning system developed in this investigation. This chapter describes how to train the system to perform certain tasks or functions. Several training examples and their simulation results are provided in this chapter.

7.2. Training the system

To train the system to perform a certain task, a teacher provides the system sequences of input/output examples that specify the task.

One requirement for training the system to perform a function or a task is as follows. To train the system to perform a repeating function, it is necessary to provide the system a sequence of input/output examples that contains one period of the function. To train the system to perform

a task that requires k steps, it is necessary to provide the system a sequence of input/output examples that contains the required k steps.

Another requirement for training the system is that all sequences of input/output examples are assumed to be starting at the same initial state. To compare an existing finite state machine with a sequence of input/output examples, the system always starts at state 0 of the finite state machine.

Five training examples are provided in the next five sections to illustrate the training and the abilities of the learning system. The training examples are: a parity checker, an up-down counter, a sequence detector, a task involving many states, and a simple code breaker.

The training examples are done on a simulation program developed and programmed by the author to test the learning system. The program is developed on an Apple PowerMac computer, using CodeWarrior C/C++ by Metrowerks Inc. The simulation program is named

Figure 7.3.1. A screen shot of the simulation program.

Archetype. A screen shot of the program is shown in Figure 7.3.1. As shown in the figure the program provides five commands to a user.

7.3. A parity checker

In this section we will train the system to perform the task of a parity checker. We will test whether the system is able to learn the task by using only one sequence of input/output examples. We specify what we mean by a parity checker by using the input/output sequence shown in Figure 7.3.2. At any step k , the parity checker is required to produce an output 1 for an odd number of 1's in the input sequence up to k . It produces an output 0, at step k , for an even number of 1's in the input sequence up to k .

Input/Output Sequence											
Step:	0	1	2	3	4	5	6	7	8	9	10
In:	0	1	0	0	1	0	1	0	1	1	0
Out:	0	1	1	1	0	0	1	1	0	1	1

Figure 7.3.2. Parity checker: I/O sequence.

Processing the input/output sequence (Figure 7.3.2), Archetype produces the state table (Figure 7.3.3) by using the process for “creating a new internal model” described in Section 4.2.

As shown in Figure 7.3.3, the process requires only 8 nodes in the minimum closed-cover search tree that is described in Section 4.2.5.

The number of nodes used by the minimum closed-cover search tree is provided for evaluating the performance of the process (described in Chapter 8).

```
(8 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
      0      1
s0 :   s0, 0   s1, 1
s1 :   s1, 1   s0, 0

New state table added.
```

Figure 7.3.3. Parity checker: resultant state table.

As demonstrated in the above simulation, the system is able to learn the task of the parity checker by using only one sequence of input/output examples that consists 11 steps.

7.4. An up-down counter

In this section we will train the system to perform the task of an up-down counter. We will test whether the system is able to learn the task by using only one sequence of input/output examples (shown in Figure 7.4.1).

This training example also shows that the system is able to handle inputs and outputs that are represented by string of characters (or multiple-bit binary numbers) having various lengths. It also shows that the system is able to take don't care characters. For example, in two of

Input/Output Sequence										
Step:	0	1	2	3	4	5	6	7	8	9
In:	up	up	up	up	down	down	down	down	down	up
Out:	One	Two	Three	Z??o	Three	Two	One	Zero	T??ee	Zero

Figure 7.4.1. Counter: I/O sequence.

(230 nodes are used by min closed-cover search tree.)		
State Table, Weight = 1		
	up	down
s0 :	s2, One	s1, Three
s1 :	s0, Zero	s3, Two
s2 :	s3, Two	s0, Zero
s3 :	s1, Three	s2, One
New state table added.		

Figure 7.4.2. Counter: resultant state table.

the outputs, i.e., “Z??o” and “T??ee”, there are unspecified or don’t care characters each of which is denoted by a “?”. Don’t care characters or don’t care bits in a binary number are useful in the design of digital circuits.

Processing the input/output sequence (Figure 7.4.1), Archetype produces the state table (Figure 7.4.2) by using the process for “creating a new internal model” described in Section 4.2.

We will now test the resultant learned model, i.e. the state table shown in Figure 7.4.2, using the test input sequence shown in row “In:” in Figure 7.4.3. The row “out1” (in Figure 7.4.3) is produced by starting at state “s0” in the state table (Figure 7.4.2). An input “down”, at state “s0”, the state table produces an output “Three”. It then goes to

From state table 0, weight = 1						
Step: 0	1	2	3	4	5	6
In: down	down	up	up	up	down	down
out1 s0, Three	s1, Two	s3, Three	s1, Zero	s0, One	s2, Zero	s0, Three

Figure 7.4.3. Counter: test result.

state “s1” at which it receives an input “down” and then produces an output “Two”, and so on, as shown in Figure 7.4.3.

As demonstrated in the above simulation, the system is able to learn the task of a four-state up-down counter by using only one sequence of input/output examples, despite some of outputs in the sequence contains unknown characters.

7.5. A sequence detector

In this section we will train the system to perform the task of a sequence detector. Four sequences of input/output examples will be used. One of the sequences contains errors or “noise” so that it is not consistent with other sequences.

Input/Output Sequence					
Step:	0	1	2	3	4
In:	0	1	1	1	0
Out:	0	0	0	1	0

Figure 7.5.1. Sequence detector: I/O sequence 1.

Suppose that we are going to design a sequence detector that detects three consecutive 1's in an input sequence and then produces an output 1, otherwise it produces an output 0. We will do so by providing examples to the system. The job of the system is to derive a model of the sequence detector from those examples.

We provide the first sequence of input/output examples as shown in Figure 7.5.1. Processing the input/output sequence (Figure 7.5.1), Archetype creates the state table (Figure 7.5.2) by using the process for “creating a new internal model” described in Section 4.2.

(4 nodes are used by min closed-cover search tree.)			
State Table, Weight = 1			
	0	1	
s0 :	s0, 0	s1, 0	
s1 :	s0, 0	s2, 0	
s2 :	s0, 0	s0,s1,s2, 1	
New state table added.			

Figure 7.5.2. Sequence detector: state table 0.

From state table 0, weight = 1						
Step:	0	1	2	3	4	5
In:	1	1	1	1	1	0
out1	s0, 0	s1, 0	s2, 1	s0, 0	s1, 0	s2, 0
out2	=	=	=	s1, 0	s2, 1	s0, 0
out3	=	=	=	=	=	s1, 0
out4	=	=	=	=	=	s2, 0
out5	=	=	=	s2, 1	s0, 0	s1, 0
out6	=	=	=	=	s1, 0	s2, 0
out7	=	=	=	=	s2, 1	s0, 0
out8	=	=	=	=	=	s1, 0
out9	=	=	=	=	=	s2, 0

Figure 7.5.3. Sequence detector: test result 1.

As shown in Figure 7.5.2, there are three possible next states “s0, s1, s2” for present state “s2” and input “1”. The reason for the three possible next states is that our examples do not provide sufficient information for the system to determine a single next state.

To see whether the model (Figure 7.5.2) meets our requirement, we provide Archetype a test input sequence that is shown in row “In:” in Figure 7.5.3. In response to the test sequence, the state table produces nine output sequences, out1 to out9 as shown in Figure 7.5.3. In the figure an “=” sign denotes that the value of that location equals the value of the one above the sign.

Input/Output Sequence										
Step:	0	1	2	3	4	5	6	7	8	9
In:	0	1	1	1	0	1	1	0	1	0
Out:	0	0	0	1	0	0	0	0	0	0

Figure 7.5.4. Sequence detector: I/O sequence 2.

To eliminate uncertainty we need more examples. We provide another set of examples as shown in Figure 7.5.4.

Using the process for “checking consistency and updating internal models” described in Section 5.2, Archetype finds that the new set of examples (Figure 7.5.4) is consistent with the existing state table (Figure 7.5.2). As shown in Figure 7.5.5, only 22 nodes are used in the search tree (described in Section 5.6) for checking consistency and updating internal models. The number of nodes used in the search tree is provided for evaluating the performance of the process (described in Chapter 8).

(22 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with state table 0

Figure 7.5.5. Sequence detector: result of sequence 2.

0 State Table, Weight = 2			
	0	1	
s0 :	s0, 0	s1, 0	
s1 :	s0, 0	s2, 0	
s2 :	s0, 0	s0,s1,s2, 1	

Figure 7.5.6. Sequence detector: state table 0 after sequence 2.

As shown in Figure 7.5.6, the state table (Figure 7.5.2) is not changed except that its weight is increased by one which is the result of the process described in Section 6.2. Although another set of examples is provided, the information contained in the examples is already there in the existing model. Thus the state table is not changed.

Now we provide another set of examples. However, the examples contain two errors. As shown in Figure 7.5.7, the output of step 3 should be an 1 not a 0. The output of step 6 should be a 0 not an 1.

Input/Output Sequence								
Step:	0	1	2	3	4	5	6	7
In:	0	1	1	1	0	1	1	0
Out:	0	0	0	0	0	0	1	0

Figure 7.5.7. Sequence detector: I/O sequence 3.

```

(4 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 0

(12 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
      0          1
s0 :      s1, 0      s2, 0
s1 :      s0, 0      s1, 0
s2 :      s0, 0      s0,s1,s2, 1

New state table built and added

```

Figure 7.5.8. Sequence detector: state table 1.

By using the process for “checking consistency and updating internal models” described in Section 5.2, Archetype detects that the new input/output sequence is not consistent with the existing state table (Figure 7.5.6). As shown in Figure 7.5.8, Archetype records the inconsistency by using the process for “handling inconsistent information” (Section 6.2).

We give another set of examples as shown in Figure 7.5.9. Using the process for “checking consistency and updating internal models” (Section 5.2), Archetype updates state table 0 (Figure 7.5.6) by

Input/Output Sequence								
Step:	0	1	2	3	4	5	6	7
In:	0	1	1	1	1	1	1	0
Out:	0	0	0	1	0	0	1	0

Figure 7.5.9. Sequence detector: I/O sequence 4.

```

(13 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with an updated version of state table 0

State Table, Weight = 3
      0      1
s0 :   s0, 0   s1, 0
s1 :   s0, 0   s2, 0
s2 :   s0, 0   s0, 1

The updated state table added.

(4 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 1

```

Figure 7.5.10. Sequence detector: state table 2.

removing possible next states “s1 , s2” from the list of possible next states associated with present state s2 and input 1. The process for removing possible next states is described in Section 5.7. The updated state table is shown in Figure 7.5.10. Archetype does not overwrite the original state table 0. It adds a new updated version of the state table.

Next it checks state table 1 (Figure 7.5.8) which is the result of the set of examples that contains two errors. It finds that the input/output sequence (Figure 7.5.9) is not consistent with the state table 1.

As the result of the above four set of examples, Archetype builds three state tables as shown in Figure 7.5.11, Figure 7.5.12, and Figure 7.5.13. State table 2 (Figure 7.5.13) has the highest weight.

0 State Table, Weight = 2		
	0	1
s0 :	s0, 0	s1, 0
s1 :	s0, 0	s2, 0
s2 :	s0, 0	s0,s1,s2, 1

Figure 7.5.11. Sequence detector: resultant state table 0.

1 State Table, Weight = 1		
	0	1
s0 :	s1, 0	s2, 0
s1 :	s0, 0	s1, 0
s2 :	s0, 0	s0,s1,s2, 1

Figure 7.5.12. Sequence detector: resultant state table 1.

2 State Table, Weight = 3		
	0	1
s0 :	s0, 0	s1, 0
s1 :	s0, 0	s2, 0
s2 :	s0, 0	s0, 1

Figure 7.5.13. Sequence detector: resultant state table 2.

We now test each of the state tables by using a test input sequence shown in row “In:” in each of the Figure 7.5.14, Figure 7.5.15, and Figure 7.5.16.

From state table 0, weight = 2						
Step:	0	1	2	3	4	5
In:	1	1	1	1	1	0
out1	s0, 0	s1, 0	s2, 1	s0, 0	s1, 0	s2, 0
out2	=	=	=	s1, 0	s2, 1	s0, 0
out3	=	=	=	=	=	s1, 0
out4	=	=	=	=	=	s2, 0
out5	=	=	=	s2, 1	s0, 0	s1, 0
out6	=	=	=	=	s1, 0	s2, 0
out7	=	=	=	=	s2, 1	s0, 0
out8	=	=	=	=	=	s1, 0
out9	=	=	=	=	=	s2, 0

Figure 7.5.14. Sequence detector: resultant test 0.

From state table 1, weight = 1						
Step:	0	1	2	3	4	5
In:	1	1	1	1	1	0
out1	s0, 0	s2, 1	s0, 0	s2, 1	s0, 0	s2, 0
out2	=	=	=	=	s1, 0	s1, 0
out3	=	=	=	=	s2, 1	s0, 0
out4	=	=	=	=	=	s1, 0
out5	=	=	=	=	=	s2, 0
out6	=	=	s1, 0	s1, 0	s1, 0	s1, 0
out7	=	=	s2, 1	s0, 0	s2, 1	s0, 0
out8	=	=	=	=	=	s1, 0
out9	=	=	=	=	=	s2, 0
out10	=	=	=	s1, 0	s1, 0	s1, 0
out11	=	=	=	s2, 1	s0, 0	s2, 0
out12	=	=	=	=	s1, 0	s1, 0
out13	=	=	=	=	s2, 1	s0, 0
out14	=	=	=	=	=	s1, 0
out15	=	=	=	=	=	s2, 0

Figure 7.5.15. Sequence detector: resultant test 1.

From state table 2, weight = 3						
Step:	0	1	2	3	4	5
In:	1	1	1	1	1	0
out1	s0, 0	s1, 0	s2, 1	s0, 0	s1, 0	s2, 0

Figure 7.5.16. Sequence detector: resultant test 2.

State table 2 has the highest weight and it produces an output sequence (Figure 7.5.16) that meets our requirement.

The preceding simulation demonstrates that the system developed in this investigation is able to accept sequences of input/output examples for the design of simple sequential machines. The system allows a designer to provide additional examples to fine tune the design. It also tolerates infrequent errors made by the designer. It is the first such a system. Other systems require design entries such as state diagrams or state tables [18], hardware description language codes [1][51], state equations and output equations [7], or Pascal-like language codes [48].

7.6. A task involving many states

In this section we will test whether the system is able to handle sequential task that requires many states. A simple task specified by the input/output sequence shown in Figure 7.6.1 that requires 30 states

Input/Output Sequence																
Step:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
In:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Out:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29			
0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0	0	0	1			

Figure 7.6.1. Many states: I/O sequence 1.

```

(31 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
0
s0 :    s1, 0
s1 :    s2, 0
s2 :    s3, 0
s3 :    s4, 0
s4 :    s5, 0
s5 :    s6, 0
s6 :    s7, 0
s7 :    s8, 0
s8 :    s9, 0
s9 :    s10, 0
s10 :   s11, 0
s11 :   s12, 0
s12 :   s13, 0
s13 :   s14, 0
s14 :   s15, 0
s15 :   s16, 0
s16 :   s17, 0
s17 :   s18, 0
s18 :   s19, 0
s19 :   s20, 0
s20 :   s21, 0
s21 :   s22, 0
s22 :   s23, 0
s23 :   s24, 0
s24 :   s25, 0
s25 :   s26, 0
s26 :   s27, 0
s27 :   s28, 0
s28 :   s29, 0
s29 :   s-1, 1

New state table added.

```

Figure 7.6.2. Many states: state table 0.

is used as an example. In addition, the simulation result of a task that requires 100 states is reported.

By using the process for “creating a new internal model” (Section 4.2), Archetype processes the examples (Figure 7.6.1) and produces a state table that has 30 states as shown in Figure 7.6.2.

We now test the process for “checking consistency and updating internal models” (Section 5.2) by using the input/output sequence shown in Figure 7.6.3.

Input/Output Sequence							
Step:	0	1	2	3	4	5	6
In:	1	1	1	0	1	0	1
Out:	0	1	0	0	1	0	0

Figure 7.6.3. Many states: I/O sequence 2.

Archetype processes the sequence (Figure 7.6.3) and produces an updated state table (Figure 7.6.4) that is consistent with the sequence.

(7470 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with an updated version of state table 0

State Table, Weight = 2

	0	1
s0 :	s1, 0	s1, 0
s1 :	s2, 0	s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15, s16,s17,s18,s19,s20,s21,s22,s23,s24,s25,s26,s27,s28, 1
s2 :	s3, 0	s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15, s16,s17,s18,s19,s20,s21,s22,s23,s24,s25,s26,s27,s28, 1
s3 :	s4, 0	s0,s1, 0
s4 :	s5, 0	s0,s1, 0
s5 :	s6, 0	s0,s1, 0
s6 :	s7, 0	s0,s1, 0
s7 :	s8, 0	s0,s1, 0
s8 :	s9, 0	s0,s1, 0
s9 :	s10, 0	s0,s1, 0
s10 :	s11, 0	s0,s1, 0
s11 :	s12, 0	s0,s1, 0
s12 :	s13, 0	s0,s1, 0
s13 :	s14, 0	s0,s1, 0
s14 :	s15, 0	s0,s1, 0
s15 :	s16, 0	s0,s1, 0
s16 :	s17, 0	s0,s1, 0
s17 :	s18, 0	s0,s1, 0
s18 :	s19, 0	s0,s1, 0
s19 :	s20, 0	s0,s1, 0
s20 :	s21, 0	s0,s1, 0
s21 :	s22, 0	s0,s1, 0
s22 :	s23, 0	s0,s1, 0
s23 :	s24, 0	s0,s1, 0
s24 :	s25, 0	s0,s1, 0
s25 :	s26, 0	s0,s1, 0
s26 :	s27, 0	s0,s1, 0
s27 :	s28, 0	s0,s1, 0
s28 :	s29, 0	s0,s1, 0
s29 :	s-1, 1	s0,s1, 0

The updated state table added.

Figure 7.6.4. Many states: state table 1.

As demonstrated in the above simulation, the system is able to handle a simple sequence task that requires 30 states.

In addition, a task that is similar to the one shown in the above simulation but it requires 100 states is tested. In this case, 101 nodes are used in the process for creating the state table, and 94690 nodes are used in the process for checking and updating internal model.

7.7. A simple code breaker

In this section we will train the system to perform the task of a simple code breaker. We will test whether the system is able to learn a combinational task.

Suppose that we want to decode an encrypted document or message, but we do not have the “key”. We are, however, able to intercept and decode three related messages. Using the three sample messages we wish to build a machine that can decode the unknown message “z 8 r 0 x b 3 7 b z” and that can help us decode other future messages.

Input/Output Sequence									
Step:	0	1	2	3	4	5	6	7	8
In:	r	7	8	0	x	r	8	7	z
Out:	P	R	O	_	S	P	O	R	T

Figure 7.7.1. Code Breaker: message 1.

We provide the first message shown in Figure 7.7.1 to Archetype. It processes the message and produces the state table shown in Figure 7.7.2. To produce the state table, the process for “creating a new internal model” described in Section 4.2 is used.

(2 nodes are used by min closed-cover search tree.)						
State Table, Weight = 1						
	r	7	8	0	x	z
s0 :	s0, P	s0, R	s0, O	s0, _	s0, S	s-1, T
New state table added.						

Figure 7.7.2. Code breaker: state table 0.

We provide the second message shown in Figure 7.7.3 to Archetype.

Input/Output Sequence									
Step:	0	1	2	3	4	5	6	7	8
In:	z	8	r	0	x	3	8	7	b
Out:	T	O	P	_	S	C	O	R	E

Figure 7.7.3. Code breaker: message 2.

Archetype checks the input/output sequence (Figure 7.7.3) against the existing internal model that is state table 0 (Figure 7.7.2). The process for “checking consistency and updating internal models” described in Section 5.2 is used. As shown in Figure 7.7.4, the result is that the input/output sequence is consistent with an updated version of the state table 0.

```
(9 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with an updated version of state table 0

State Table, Weight = 2
      r      7      8      0      x      z      3      b
s0 :   s0, P   s0, R   s0, O   s0, _   s0, S   s0, T   s0, C   s-1, E
The updated state table added.
```

Figure 7.7.4. Code breaker: state table 1.

Compare the updated version (Figure 7.7.4) with the original version (Figure 7.7.2) produced earlier, we can see that two new input columns are added, i.e. the columns for inputs 3 and b that are located at the end of the updated state table (Figure 7.7.4). The process for adding new input column is described in Section 5.3. We can also see that the next

state for the input z has been changed from unspecified next state “s-1” to a specified next state “s0”.

We provide the third message shown in Figure 7.7.5 to Archetype.

Input/Output Sequence								
Step:	0	1	2	3	4	5	6	7
In:	x	b	z	0	3	r	x	z
Out:	S	E	T	—	K	O	S	T

Figure 7.7.5. Code breaker: message 3.

Archetype checks the input/output sequence (Figure 7.7.5) against the two existing internal models, state tables 0 (Figure 7.7.2) and state table 1 (Figure 7.7.4). As shown in Figure 7.7.6, the result is that the input/output sequence is not consistent with any existing models. Archetype records such an inconsistency by adding a new internal model as shown in Figure 7.7.6. The process for “handling inconsistent information” described in Section 6.2 is used.

```

(6 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 0

(5 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 1

(2 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
      x      b      z      0      3      r
s0 :   s0, S   s0, E   s0, T   s0, _   s0, K   s0, O

New state table built and added

```

Figure 7.7.6. Code breaker: state table 2

As the result of the training, three state tables are built as shown in Figure 7.7.7, Figure 7.7.8, and Figure 7.7.9.

```

0 State Table, Weight = 1
      r      7      8      0      x      z
s0 :   s0, P   s0, R   s0, O   s0, _   s0, S   s-1, T

```

Figure 7.7.7. Code breaker: resultant state table 0.

```

1 State Table, Weight = 2
      r      7      8      0      x      z      3      b
s0 :   s0, P   s0, R   s0, O   s0, _   s0, S   s0, T   s0, C   s-1, E

```

Figure 7.7.8. Code breaker: resultant state table 1.

```

2 State Table, Weight = 1
      x      b      z      0      3      r
s0 :   s0, S   s0, E   s0, T   s0, _   s0, K   s0, O

```

Figure 7.7.9. Code breaker: resultant state table 2.

To see why the last input/output sequence (Figure 7.7.5) is not consistent with state table 0 and 1, we will compare the input/output sequence with the output sequences provided by the state tables. As shown in Figure 7.7.10, the given input sequence is shown in row “In:”, while the required output sequence is shown in row “Out:”. The output sequence produced by a state table is shown in row “out1”.

The process for comparing two output sequences is described in Section 5.5. The input/output sequence is not consistent with state table 0 (Figure 7.7.10), because in step 5 the required output “O” is not equal to the produced output “P”. Similarly, the input/output sequence is not consistent with state table 1 (Figure 7.7.11), because the required outputs in steps 4 and 5 are not equal to the corresponding produced outputs. Finally, the input/output sequence is consistent with state table 2 (Figure 7.7.12) which is the newly created state table based on the sequence.

From state table 0, weight = 1								
Step:	0	1	2	3	4	5	6	7
In:	x	b	z	0	3	r	x	z
Out:	S	E	T	_	K	O	S	T
out1	s0, S	s0, *	s0, T	s0, _	s0, *	s0, P	s0, S	s0, T

Figure 7.7.10. Code breaker: comparison 0.

From state table 1, weight = 2								
Step:	0	1	2	3	4	5	6	7
In:	x	b	z	0	3	r	x	z
Out:	S	E	T	_	K	O	S	T
out1	s0, S	s0, E	s0, T	s0, _	s0, C	s0, P	s0, S	s0, T

Figure 7.7.11. Code breaker: comparison 1.

From state table 2, weight = 1								
Step:	0	1	2	3	4	5	6	7
In:	x	b	z	0	3	r	x	z
Out:	S	E	T	_	K	O	S	T
out1	s0, S	s0, E	s0, T	s0, _	s0, K	s0, O	s0, S	s0, T

Figure 7.7.12. Code breaker: comparison 2.

We will now check whether the learned models (Figure 7.7.7, Figure 7.7.8, and Figure 7.7.9) are able to decode the unknown message “z 8 r 0 x b 3 7 b z”.

We provide the unknown message as test input sequence to Archetype. It processes the test input sequence and produces the printouts shown in Figure 7.7.13, Figure 7.7.14, and Figure 7.7.15. The row “out1”

from state table 0 (Figure 7.7.13), for example, is produced by starting at state “s0”. Given an input “z”, the state table produces an output “T”. Then it goes to state “s0”. Given an input “8”, it produces and output “O”, and so on. An output “*” is an unspecified or unknown output.

From state table 0, weight = 1										
Step: 0	1	2	3	4	5	6	7	8	9	
In: z	8	r	0	x	b	3	7	b	z	
out1	s0, T	s0, O	s0, P	s0, _	s0, S	s0, *	s0, *	s0, R	s0, *	s0, T

Figure 7.7.13. Code breaker: test result 0.

From state table 1, weight = 2										
Step: 0	1	2	3	4	5	6	7	8	9	
In: z	8	r	0	x	b	3	7	b	z	
out1	s0, T	s0, O	s0, P	s0, _	s0, S	s0, E	s0, C	s0, R	s0, E	s0, T

Figure 7.7.14. Code breaker: test result 1.

From state table 2, weight = 1										
Step: 0	1	2	3	4	5	6	7	8	9	
In: z	8	r	0	x	b	3	7	b	z	
out1	s0, T	s0, *	s0, O	s0, _	s0, S	s0, E	s0, K	s0, *	s0, E	s0, T

Figure 7.7.15. Code breaker: test result 2.

State table 1 (Figure 7.7.14) has the highest weight. It produces the output sequence “T O P _ S E C R E T” which is the most probable message based on the resultant learned models.

As demonstrated in the above simulation, the system developed in this investigation is able to learn combinational task that is represented by a state table that has only one present state.

8. PERFORMANCE

8.1. Introduction

This chapter discusses the performance of the learning system. It provides the results of evaluating the performance of the process for creating a new internal model, and the process for checking consistency and updating internal model. It also describes methods implemented in the learning system for improving the performance.

8.2. Performance of the process for creating a new internal model

The performance of the process for creating a new internal model is limited by the step for finding a minimum closed cover (described in Section 4.2.5) which has a search space having an upper bound of $(1 - p^{m+1})/(1 - p)$ nodes, where p is the number of prime compatibility classes, and m is the number of classes in the minimum closed cover.

To reduce the search space, the method implemented in this investigation is to skip the process of deriving all prime compatibility classes (Section 4.2.4). The set of all maximum compatibility classes (found in Section 4.2.3) is then used as the prime compatibility classes for finding a minimum closed cover. Since the set of all prime compatibility classes is usually much larger than the set of all maximum compatibility classes, the solution greatly reduces the search space.

The solution does not downgrade the learning ability. In most tests the results are the same whether the process of deriving all prime compatibility classes (Section 4.2.4) is skipped or not. In other tests the process must be skipped. Thus, in version 1.0 β of the simulation program Archetype, the process is permanently skipped.

Name of application examples	Number of steps in the input/output sequence	Number of states in the generated state table	Number of nodes used by the minimum closed-cover search tree
An up-down counter	10	4	230
A parity checker	11	2	8
A sequence detector	5	3	4
	8	3	12
A task involving many states	30	30	31
	100	100	101
A simple code breaker	9	1	2
	8	1	2

Figure 8.2.1. Summary of simulation results for creating new internal models.

The simulation results reported in Chapter 7 is produced by using Archetype 1.0β. The number of nodes used by the minimum closed-cover search trees are shown in Figure 8.2.1.

8.3. Performance of the process for checking consistency and updating internal models

The process for checking consistency and updating internal models (described in Section 5.2) has a search space having an upper bound of n^k nodes, where n is the number of states in a state table, and k is the

number of steps in a given input/output sequence. Two methods are used to reduce the number of nodes needed to be checked.

The first method is to use the depth-first search process described in Section 5.6. As shown in Figure 8.3.1, nodes that are denoted by “=”s do not need to be generated or compared. An “=” sign denotes that the value of that location equals the value of the one above the sign. Output sequences are denoted by “out1” to “out9”.

Step:	0	1	2	3	4	5
out1	s0, 0	s1, 0	s2, 1	s0, 0	s1, 0	s2, 0
out2	=	=	=	s1, 0	s2, 1	s0, 0
out3	=	=	=	=	=	s1, 0
out4	=	=	=	=	=	s2, 0
out5	=	=	=	s2, 1	s0, 0	s1, 0
out6	=	=	=	=	s1, 0	s2, 0
out7	=	=	=	=	s2, 1	s0, 0
out8	=	=	=	=	=	s1, 0
out9	=	=	=	=	=	s2, 0

Figure 8.3.1. Reduce search space by depth-first search process.

As shown in Figure 8.3.1, the number of nodes needed to be checked reduce to 20 from the total of 54.

The depth-first search process is stopped as soon as an inconsistency is found. Thus, not all the output sequences are required to be generated or compared.

The second method to reduce the number of nodes needed to be checked is incorporated in the process for removing possible next states (described in Section 5.7). Removing a possible next state corresponding to an output sequence will remove the possibility for generating the output sequence. Since a next state is usually responsible for generating many output sequences, removing one next state will prevent many other corresponding output sequences from being generated.

Name of application examples	Number of steps in the input/output sequence	Number of states in the state table used for checking	Number of nodes used by the process for checking and updating internal model
A sequence detector	10	3	22
	8	3	4
	8	3	13
	8	3	4
A task involving many states	7	30	7470
	7	100	94690
A simple code breaker	9	1	9
	8	1	6
	8	1	5

Figure 8.3.2. Summary of simulation results for checking and updating internal models.

The numbers of nodes needed to be checked for the application examples described in Chapter 7 are summarized as shown in Figure 8.3.2.

9. SUMMARY AND CONCLUSION

A system that is capable of learning both sequential and combinational tasks from examples is described in detail in this dissertation.

The system learns from sequences of input/output examples provided by a teacher or another system. It does not ask questions and does not perform experiments on its environment. It learns how to perform a task by observing sequences of steps that specify the task. From the sequences of steps, it builds finite state machines to describe the task.

When additional sequences of input/output examples are available, the system is capable of modifying its finite state machines to accommodate the new information. It is able to expand or to limit the scope of its finite state machines to conform to the new information and to preserve its old information as well.

The system can detect any inconsistency that may result from noise by comparing the sequences of input/output examples with its finite state

machines. It builds new finite state machines to record any inconsistency or any abrupt change. It also evaluates each of its finite state machines by assigning a weight for each of the machines.

New algorithms are developed and new concepts are defined by the author in order to make such a system possible. A simulation program is developed and coded by the author to test the system. In addition, several application examples are provided.

The system developed in this investigation is often capable of inferring a reasonable finite state machine from just one sequence of input/output examples. On the contrary, recurrent neural networks often require thousands of sequences. On the other hand, recurrent neural networks should perform better in noisy environment than the system developed in this investigation.

The system developed in this investigation uses finite state machines to represent learned models. The resultant learned models are often deterministic finite state machines that are represented by using state tables. Moreover, the finite state machines often contain minimal

number of states. Thus, the resultant learned models can easily and economically be implemented in VLSI. On the contrary, neural network models are more difficult to be implemented in hardware.

The system developed in this investigation incorporates new information into existing state tables using three methods: adding new input columns, removing possible next states, and specifying don't cares. One future extension of the system is to develop a method to allow the system to add more states into existing state tables. This may remove the requirement that to train the system to perform a repeating function, it is necessary to provide the system a sequence of input/output examples that contains one period of the function.

10. REFERENCES

- [1] Amellal, S. and Kaminska, B., "Functional Synthesis of Digital Systems with TASS", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 5, 537-552, 1994.
- [2] Anderson, Howard C. , "Neural Network Machines", *IEEE Potentials*, pp. 13-16, February, 1989.
- [3] Angluin, Dana, "A Note on the Number of Queries Needed to Identify Regular Languages", *Information and Control*, 1981.
- [4] Angluin, Dana, "Learning Regular Sets from Queries and Counter examples", *Information and Computation*, 1987.
- [5] Angluin, Danna and Smith, Carl H., "Inductive Inference: Theory and Methods," *Computing Surveys*, Vol. 15, No. 3., pp. 237-269, 1983.
- [6] Barto, Andrew C. and P. Anandan, "Pattern-Recognizing Stochastic Learning Automata," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-15, No.3, pp. 360-375, May/June 1985.
- [7] Bekki, K, Nagai, T., Hamada, N, Shimizu, T., Hiratsuka, N., and Shima, K., "An Automatic Finite State Machine Synthesis Using Temporal Logic Decomposition", *IEEE Conference on Computer-aided Design (ICCAD)*, pp. 422-425, 1991.
- [8] Bengio, Yoshua, Paolo Frasconi, and Patrice Simard, "The Problem of Learning Long-term Dependencies in Recurrent Networks," *IEEE International Conference on Neural Networks*, 1993.
- [9] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi, "Learning Long-term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, Vol. 5, No. 2, March 1994.

- [10] Bianchini, M., M. Gori, and M. Maggini, "On the Problem of Local Minima in Recurrent Neural Networks," *IEEE Transactions on Neural Networks*, Vol. 5, No. 2, March 1994.
- [11] Bibyk, Steven, and Mohammed Ismail, "Issues in Analog VLSI and MOS Techniques for Neural Computing," *Analog VLSI Implementation of Neural systems*, edited by Carver Mead and Mohammed Ismail, pp.103-133, 1989.
- [12] Biermann, A. W. and Feldman, J. A., "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, pp. 592-597, June 1972.
- [13] Breeding, Kenneth J., in communication with, Department of Electrical Engineering, The Ohio State University.
- [14] Carbonell, J. (edited), *Machine Learning: Paradigms and Methods*, The MIT Press, 1990.
- [15] Carroll, John, and Darrel Long, *Theory of finite automata with an introduction to formal languages*, p. 13, p. 119, Prentice-Hall, Inc., 1989.
- [16] Chen, Han-Fu and Lei Guo, *Identification and Stochastic Adaptive Control*, Birkhauser, 1991.
- [17] Cohen, Paul R. and Edward A. Feigenbaum, edited , *The Handbook of Artificial Intelligence*, Vol. III., Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1982.
- [18] Ebata, K., Kutsuwa, T., Kadomaru, N., "Automatic Synthesis System of Sequential Circuits," *IEEE International Symposium of Circuits and Systems*, pp. 969-972, 1988.
- [19] Fu, K. S., *Syntactic Methods in Pattern Recognition*, Academic Press, pp. 212-217, 1974.
- [20] Giles, C. L., G. M. Kuhn, and R. J. Williams, Guest Editors, Special Issue on Dynamic Recurrent Neural Networks, *IEEE Transactions on Neural Networks*, Vol. 5, No 2, March 1994.
- [21] Gold, E. Mark, "System Identification via State Characterization", *Automatica*, 1972.

- [22] Gold, E. Mark, "Complexity of Automaton Identification from Given Data," *Information and Control*, 1978.
- [23] Gold, E. Mark, "Language Identification in the Limit," *Information and Control*, 1967.
- [24] Grasselli, A. and Luccio, F., "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," *IEEE Transactions on Electronic Computers*, pp. 350-359, 1965.
- [25] Hecht-Nielsen, Robert, *Neurocomputing*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.
- [26] Hootman, Joe, Ed.-in-Chief, *IEEE Micro: Special Issue on Silicon Neural Networks*, December, 1989.
- [27] Ibarra, Osca H. and Tao Jiang, "Learning Regular Languages from Counterexamples", *Proceedings of the 1988 Workshop on Computational Learning Theory*, 1988.
- [28] Ismail, Mohammed, and Terri Fiez, *Analog VLSI Signal and Information Processing*, McGraw-Hill, Inc., 1994.
- [29] Jordan, Michael I. "Attractor dynamics and parallelism in connectionist sequential machine," *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pp. 521-545, 1986.
- [30] Kalaba, Robert and Karl Spingarn, *Control, Identification, and Input Optimization*, Plenum Press, 1982.
- [31] Khanna, Tarun, *Foundations of neural Networks*, Addison-Wesley Publishing Company, Inc., 1990.
- [32] Kohavi, Zvi, *Switching and Finite Automata Theory*, McGraw-Hill Book Company, p. 323, pp.333-347, pp.458-464, 1978.
- [33] Lancotot, Kevin, and B. John Oommen, "Discretized Estimator Learning Automata," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 22, No. 6, November/December 1992.
- [34] Larsen, J. Richard and Morris L. Marx, *An Introduction to Mathematical Statistics and Its Applications*, Prentice-Hall, 1986.

- [35] Litteral, Larry A., "An Algorithm for Solving the Sequential Machine Identification Problem", Thesis, The Ohio State University, 1973.
- [36] Marron, Assaf, "Learning Pattern Languages form a Single Initial Example and from Queries", Proceedings of the 1988 Workshop on Computational Learning Theory, 1988.
- [37] McClelland, James L. and David E. Rumelhart, *Explorations in Parallel Distributed Processing: A handbook on Models, Programs, and Exercises*, The MIT Press, Cambridge, Massachusetts, 1988.
- [38] McClelland, James L., David E. Rumelhart, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition Volume 2: Psychological and Biological Models*, The MIT Press, Cambridge, Massachusetts, 1987.
- [39] Mead, Carver, *Analog VLSI and Neural Systems*, Addison-Wesley Publishing Company, 1989.
- [40] Mead, Carver, and Mohammed Ismail, *Analog VLSI implementation of neural systems*, Kluwer Academic Publishers, Boston, 1989b.
- [41] Meisel, W. S., "A Note on Internal State Minimization in Incompletely Specified Sequential Networks," *IEEE Transactions on Electronic Computers*, pp. 508-509, 1967.
- [42] Michalski, R. and Tecuci, G. (edited), *Machine Learning: A Multistrategy Approach*, Volume IV, Morgan Kaufmann Publishers, 1994.
- [43] Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann Publishers, Inc., (p.9), 1983.
- [44] Mozer, Michael C. and Jonathan Bachrach, "SLUG: A Connectionist Architecture for Inferring the Structure of Finite-State Environments," *Machine Learning*, 7, 139-160, 1991.
- [45] Narendra, Kumpatic and M.A.L. Thathachar, *Learning Automata: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.
- [46] Nilsson, Nils J., *The Mathematical Foundations of Learning Machines*, Morgan Kaufmann Publishers, San Mateo, California, pp.1-13, 1990.

- [47] Oommen, B. John; Andrade, Nite; Iyengar, S. Sitharam, "Trajectory planning of robot manipulators in noisy work spaces using stochastic automata," *International Journal of Robotics Research* v 10 n 2, pp. 135-148, 1991.
- [48] Peng, Z. and Kuchinski, K., "Automated Transformation of Algorithms into Register-Transfer Level Implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 2, 150-165, 1994.
- [49] Pollack, J.B., "Connectionism: past, present, and future", *Artificial Intelligence Review* 3, pp. 3-22, 1989.
- [50] Porat, Sara and Jerome A. Feldman, "Learning Automata from Ordered Examples", *Proceedings of the 1988 Workshop on Computational Learning Theory*, 1988, also on *Machine Learning*, 1991.
- [51] Rajan, S., "Practical State Machine Design Using VHDL," *Integrated System Design*, February 1995, pp.58-70, 1995.
- [52] Rich, Elaine, *Artificial Intelligence*, McGraw-Hill Book Company, 1983.
- [53] Rivest, Ronald L. Schapire, Robert E., "Inference of finite automata using homing sequences," *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, p 411-420, May 15-17, 1989.
- [54] Roth, Michael W., "Survey of Neural Network Technology for Automatic Target Recognition," *IEEE Transactions on Neural Networks*, Vol. 1, No.1, pp. 28-43, March, 1990.
- [55] Rouvellou, Isabelle and George W. Hart, "Inference of a probabilistic finite state machine from its output", *IEEE transactions on systems, man, and cybernetics*, pp. 424-437, vol. 25, no. 3, March 1995.
- [56] Rumelhart, D. E., G. E. Hinton, and J. L. McClelland, "A General Framework for Parallel Distributed Processing", *Parallel Distributed Processing*, Vol. 1, pp. 45-76, 1987.
- [57] Rumelhart, D. E., G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation", *Parallel Distributed Processing*, Vol. 1, pp. 318-364, 1987.

- [58] Rumelhart, David E., James L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition Volume 1: Foundations*, The MIT Press, Cambridge, Massachusetts, 1987.
- [59] Schapire, Robert Elias, "Diversity-Based Inference of Finite Automata", MIT/LCS/TR-414, 1988.
- [60] Sinha, N. K. and B. Kuszta, *Modeling and Identification of Dynamic Systems*, Van Nostrand Reinhold Company, 1983.
- [61] Soderstrom, Torsten and Petre Stoica, *System Identification*, Prentice Hall, 1989.
- [62] Tsetlin, M. L., *Automaton Theory and Modeling of Biological Systems*, Academic Press, 1973.